



M255 Unit 6

UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Subclassing and inheritance

Unit 6

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at <http://www.open.ac.uk> where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit <http://www.ouw.co.uk>, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Copyright © 2006, 2008 The Open University.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd of 90 Tottenham Court Road, London, W1T 4LP.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5498 8

2.1

CONTENTS

Introduction	5
1 Exploring the inheritance hierarchy for the amphibian classes	6
1.1 Why subclasses?	6
1.2 Exploring the behaviour of the Frog and HoverFrog classes	8
1.3 The Object class and indirect inheritance	10
1.4 How the HoverFrog class is implemented	11
1.5 Access modifiers in Java	17
2 Constructors	19
2.1 Implementation of constructors and the role of <code>super()</code>	19
2.2 Constructors and overloading	22
3 A subclass for Account	25
3.1 Creating a new class in BlueJ	25
3.2 Declaring instance variables and defining accessor methods	26
3.3 Writing the constructors	28
3.4 Modifying the behaviour of the CurrentAccount class	30
3.5 Additional behaviour for CurrentAccount	32
4 Revisiting the amphibian hierarchy	37
4.1 Capturing common behaviour: abstract classes	38
5 Subclassing, subtypes and substitution	42
6 An introduction to interfaces	47
6.1 Creating and implementing an interface type	48
6.2 Using an interface type	50
7 Summary	54
Glossary	56
Index	59

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Composer

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

This unit builds on your study of Block I, where you were introduced to the important ideas of object-oriented programming and where you began to learn about their implementation in Java. In particular, the unit aims to extend your understanding of inheritance and further explores the creation of new instances of a class and the way in which these are initialised using constructors. We start by investigating an existing class hierarchy – the various amphibian classes – and go on to guide you through the design and implementation of a new class of your own. The practical work in the unit aims to develop the programming skills you acquired in *Units 4* and *5*. It will give you your first experience of writing a Java class from scratch and will provide an important foundation for the programming that you will do for the remainder of the course. We complete the unit by discussing some of the theoretical ideas associated with inheritance, and their implications for programming in Java. The unit also introduces you to Java interfaces.

For a large part of this unit you will need to use the BlueJ IDE in conjunction with the course text. It would therefore be helpful if you could study the unit while you have access to your computer. It is very important that you carry out the activities, because most of them involve aspects of programming that you will be using throughout the remainder of the course. Section 3, which involves a lot of practical work, is likely to take more time than other sections.

1

Exploring the inheritance hierarchy for the amphibian classes

We start this section by reviewing the idea of **classification**, and its advantages for programming. You will then revisit the relationship between the `Frog` and `HoverFrog` classes, and learn how they fit into the class hierarchy to which all Java classes belong. Finally you will study the Java code for the `HoverFrog` class, paying particular attention to those aspects specifically concerned with inheritance.

1.1 Why subclasses?

Software objects can also model roles, events, and so on, but we are mainly concerned with those that represent tangible entities.

In an object-oriented approach to programming, a class of software objects often models the behaviour of an identifiable set of real-world objects and the attributes these objects need in order to carry out that behaviour. Although our `Frog` objects are not exactly real-world objects, they have clearly defined behaviour, which involves changing their position and colour; the attributes they need to carry out this behaviour are `colour` and `position`. The choice of the word ‘class’ to describe a collection of objects with similar structure and behaviour is no accident. It mirrors the situation where real-world entities are classified into different categories depending on their characteristics and behaviour. Classification helps us to manage the complexity of the world by focusing on the common characteristics of objects and situations. Red, green and blue are all in the category ‘colour’; M150, M255 and M256 are all in the category ‘Open University course’.

Exercise 1

Suggest a category for each of the following sets of entities.

- (a) Car, bus, cycle, motorcycle
- (b) Whale, cow, pig
- (c) Worm, octopus, insect.

Solution

- (a) Vehicle or form of transport
- (b) Animal or mammal
- (c) Animal or invertebrate.

Your answers might have been different, but equally correct.

There are a number of websites which provide more details on the way in which animals are classified. If you are interested you might like to try searching on ‘animals’ and ‘classification’.

Exercise 1 shows that it is possible for an object to be in more than one category (or class). Our second suggestion for each of parts (b) and (c) provides a more specialised class than our first. An octopus is an animal, but it is also an invertebrate (an animal without a backbone); in fact, it’s a member of an even more specialised class, mollusc. Figure 1 contains a very small part of the standard classification of animals and shows how the animals in Exercise 1 fit in.

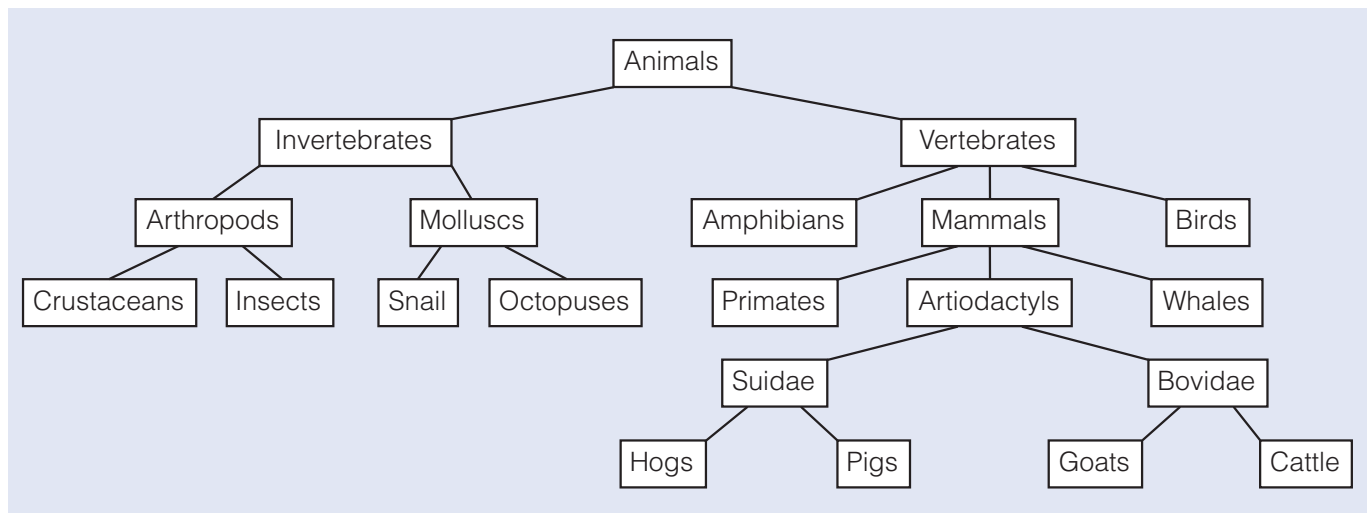


Figure 1 Extract from the classification of animals (note that most categories in the figure are incomplete in terms of their membership)

Members of the categories lower down the classification (the subclasses) have all the defining characteristics of those in the categories above them – we can take these as given – as well as some distinctive characteristics and behaviour of their own. So, for example, all mammals feed their young with their own milk, but only artiodactyls are ‘even-toed’ and of those only bovines have a certain kind of horn. A **subclass** both extends and specialises the behaviour of its superclass.

Our discussion so far has focused on the classification of already existing entities. The approach taken to software classes has similarities with this process, but it is not exactly analogous. In creating new software classes, we are extending the class hierarchy as we go, by defining each new class as an extension of some existing class. So for example having decided on the need for a class to represent hoverfrogs, we analyse the attributes and behaviour required of `HoverFrog` objects, so as to decide how best to incorporate a `HoverFrog` class into the existing class hierarchy.

As you have seen, instances of the `HoverFrog` class have both the attributes of the `Frog` class (`position` and `colour`) and behave in a way that is, in many respects, the same as that of `Frog` instances. Indeed they exhibit *all* the behaviour exhibited by `Frog` objects, but they also have additional, specialised behaviour: they can move up and down. As discussed in *Unit 2*, it is appropriate therefore to think of our mythical hoverfrogs as a subclass of frogs. Java, in company with other object-oriented programming languages, provides facilities for defining a new class as a subclass of an existing class, and this is what the course team has done in the case of the `HoverFrog` and `Frog` classes.

The conceptual advantage of **subclassing** – the process whereby a new class is defined as a specialised type of some existing class – is that it enables us to reduce complexity by focusing only on the distinctive characteristics of the more specialised class. In a programming context there are also practical advantages. The most important of these is that we are able make use of the work already done in implementing the behaviour of the superclass. As you have seen, the methods (behaviour) of one class are automatically available to (we say **inherited** by) its subclass(es). Not only does this save us the work involved in redesigning and rewriting program code, but it also reduces the amount of testing needed and minimises the likelihood of errors. Furthermore, if changes are needed in an aspect of the behaviour shared by a class and its subclass(es), only one class needs to be modified.

Up until this point we have carefully and correctly been referring to classes with a full description, for example ‘the `HoverFrog`’ class. This can become very tedious, so we sometimes shorten it to the class name alone, as in ‘the method for `HoverFrog`’. Though this shorthand is commonly used, you should be aware that it is not strictly correct.

1.2 Exploring the behaviour of the Frog and HoverFrog classes

We will begin our study of the amphibian class hierarchy by revisiting the attributes and behaviour of `HoverFrog` objects and reminding ourselves of the way in which these relate to the attributes and behaviour of `Frog` objects.

SAQ 1

Here is a list of the messages we have defined for the `Frog` class: `brown()`, `croak()`, `getColour()`, `getPosition()`, `green()`, `home()`, `jump()`, `left()`, `right()`, `sameColourAs()`, `setColour()`, `setPosition()`, `toString()`. As you learnt in *Unit 2*, the `HoverFrog` class has all these messages in its protocol too. In addition, `HoverFrog` objects can respond to the following messages: `down()`, `downBy()`, `getHeight()`, `setHeight()`, `up()`, `upBy()`.

- Which messages in the protocol of the `HoverFrog` class have the same names as those for the `Frog` class and result in identical behaviour when sent to an instance of either class?
- Are there any messages which when sent to an instance of `HoverFrog` result in behaviour that is different from when the message is sent to an instance of `Frog`? If so, which messages are they?
- From what you have learnt so far in the course, and from your solutions to parts (a) and (b), write down a list of the methods that you think will need to be defined for the `HoverFrog` class, and a list of additional attribute(s) that will be needed by the `HoverFrog` class in order that the new behaviour can be provided.

ANSWER.....

- The following messages have the same names, and result in the same behaviour, for both classes: `brown()`, `croak()`, `getColour()`, `getPosition()`, `green()`, `jump()`, `left()`, `right()`, `sameColourAs()`, `setColour()`, `setPosition()`.
- `HoverFrog` instances behave slightly differently from `Frog` instances in response to the messages `home()` and `toString()`. In addition to returning to their original position, `HoverFrog` instances will respond to a message `home()` by reverting to their original height of 0, and in response to a message `toString()` they will return a textual representation which includes details of their height.
- The methods that need to be defined for `HoverFrog` include all those that provide new or modified behaviour, namely: `down()`, `downBy()`, `getHeight()`, `home()`, `setHeight()`, `toString()`, `up()` and `upBy()`. The additional instance variable needed in order to implement the new and changed behaviour is `height`.

It is a moot point whether instances of the `HoverFrog` class behave in the same way as instances of the `Frog` class in response to a message `jump()`, as hoverfrogs cannot jump once they are 'above a stone' (though they jump identically to frogs from height zero). For the purposes of our discussion we will take the view that both respond in the same way.

As a result of SAQ 1 we conclude that several new methods are needed for the `HoverFrog` class, and that two methods may need some modification. We now look at the specification for the `Frog` and `HoverFrog` classes.

ACTIVITY 1

Launch BlueJ and open the project `Unit6_Project_1`. In the main BlueJ window, select Project Documentation from the Tools menu to generate the **Javadoc** documentation for the project. When your web browser opens with the Javadoc documentation, click on the `HoverFrog` link in the All classes frame at the left of the window. This results in the documentation for the `HoverFrog` class being displayed in the main part of the window.

Scroll down to the Method Summary for the class. Read through this section and make a note of the methods listed.

Now click on the Frog link in the All classes pane, navigate to the Method Summary and briefly read through the list of methods for that class.

DISCUSSION OF ACTIVITY 1

The Method Summary for HoverFrog is reproduced in Figure 2.

Method Summary	
void	down () If the height of the receiver is greater than 0 decrements the height of the receiver by 1, otherwise height remains unchanged.
void	downBy (int stepChange) Decreases the height of the receiver by the value of the argument stepChange.
int	getHeight () Returns the height of the receiver.
void	home () Resets the receiver to its home position and to a height of 0.
void	setHeight (int aHeight) Sets the height of the receiver to the value of the argument aHeight.
String	toString () Returns a string representation of the receiver.
void	up () If the height of the receiver is less than 6 increments the height of the receiver by 1, otherwise height remains unchanged.
void	upBy (int stepChange) Increases the height of the receiver by the value of the argument stepChange.

Figure 2 Javadoc Method Summary for the `HoverFrog` class

The **Method Summary** contains brief descriptions of all the methods corresponding to the messages that were identified in part (c) of SAQ 1. Because Javadoc is meant to provide information for other programmers wishing to use the class, private features are not listed. As the only new instance variable has been defined as private, no list of **fields** (instance variables) is shown as the documentation only shows public instance variables. You can think of Javadoc as providing the 'public face' of a class. It shows all the information needed by a programmer wishing to use the class, but reveals none of the internal details.

Instance variables are sometimes referred to in Java as **data members**, **data fields** or, as in Javadoc, **fields**. The terms 'data member' and 'data field' are slightly broader in meaning, as they include data items which are not instance variables. You will learn about these later in the course.

The Javadoc documentation confirms that when defining a new class as a subclass of an existing class, the only new methods that need to be provided are those that implement new or different behaviour. The only instance variables that need to be defined are those necessary to support that behaviour. In the next subsection you will be looking at the Java code needed to define the `HoverFrog` class as a subclass of the `Frog` class, but before doing that we will return briefly to the Javadoc information provided for the `HoverFrog` class, and introduce you to another very important class.

1.3 The Object class and indirect inheritance

When you first looked at documentation for the `HoverFrog` class in Activity 1 you may have noticed the following diagram at the very top of the window.



Figure 3 Inheritance hierarchy for `HoverFrog`, as shown in the Javadoc documentation

Where the classes are shown as hyperlinks their documentation can be accessed by clicking on the link.

Generally, if we just say that a class is a subclass of another, we mean that it is a direct subclass.

The class `OUAnimatedObject` has been written by the M255 Course Team to provide amphibian and shape objects with graphical representations.

If you are interested, the documentation for `OUAnimatedObject` and other classes in the OU class library can be obtained by selecting OU Class Library from the Tools menu in the main BlueJ window (as you did in Unit 4).

This diagram shows the complete inheritance hierarchy (or tree) for the class, including classes provided by Java. Each L-shaped link indicates that the class below has been implemented as a **direct subclass** of the class above it. So the `HoverFrog` class is shown as inheriting directly from the `Frog` class. At the very top of the tree, is a class called `Object`. The information preceding the class name tells the reader in which library the class can be found: it is defined in the main Java language library. In fact, `Object` is the progenitor (ancestor) of all other classes in Java. What the diagram as a whole tells us is that:

- ▶ a class called `Observable` has been implemented as a direct subclass of `Object`;
- ▶ a class called `OUAnimatedObject` has then been implemented as a direct subclass of `Observable`;
- ▶ the `Frog` class has been implemented as a direct subclass of `OUAnimatedObject` (`OUAnimatedObject` is the **direct superclass** of `Frog`);
- ▶ the `HoverFrog` class has been implemented as a direct subclass of `Frog` (`Frog` is a direct superclass of `HoverFrog`).

The corollary of the above information is that `Frog` is an **indirect subclass** of both `Object` and `Observable` while `HoverFrog` is an indirect subclass of `Object`, `Observable` and `OUAnimatedObject`. Every class in Java (with the exception of `Object` itself) is either a direct or an indirect subclass of `Object`. Conversely, `Object` is a direct or **indirect superclass** of every other Java class.

As you would expect, a class inherits characteristics – both attributes and behaviour – from *all* its superclasses, both direct and indirect. The `Object` class provides behaviour that is common to all Java classes. If you look at the section following the Method Summary for the `HoverFrog` class in the Javadoc documentation you will see a list of the methods implementing this behaviour, as well as a list of the methods inherited from `Frog`. The methods from the `Object` class may not mean very much to you at present, but we will be making use of some of them later in the unit.

You can now close down your browser (though you may like to explore the documentation a little further to see what it provides). If you are planning to study the next subsection immediately, you should leave BlueJ running.

1.4 How the HoverFrog class is implemented

ACTIVITY 2

If necessary launch BlueJ and open Unit6_Project_2. Double-click on the HoverFrog icon to look at its code in the editor. Scroll slowly through the class to get an overview of what it contains.

DISCUSSION OF ACTIVITY 2

You will see that the code declares the additional instance variable, `height`, and all the methods listed in the Method Summary that we looked at in the previous subsection. You may also have noticed that all these methods have the access modifier `public`. The class also has a header and a constructor.

As this subsection is concerned with the techniques for implementing a subclass in Java, the following discussion will focus mainly on the parts of the class definition that are particularly relevant to programming for inheritance. Although we will be reproducing all the relevant code in the course text, you may want to keep the class's definition open in the BlueJ editor, so that you can see how it all fits together.

In Java, to specify that some new class is to be a subclass of some existing class, the **class header** includes the keyword `extends` followed by the superclass's name. The following line declares `HoverFrog` as a publicly available class, that extends `Frog`, i.e. it is a subclass of `Frog`.

```
public class HoverFrog extends Frog
```

This line precedes the first opening brace. The first thing you would normally find inside the braces for any class definition is the declaration of the variables for the class, excluding any that have already been declared in its superclass(es).

For the class `HoverFrog` one additional instance variable, `height`, of the primitive type `int` is declared.

```
private int height;
```

In any class definition, the declaration of variables is followed by the constructor (or constructors). We will defer our discussion of constructors until Section 2, when we explore their role and implementation in some detail. Next come the method definitions for the class. For the `HoverFrog` class a number of methods are required to deal with the fact that hoverfrogs can change their height: `getHeight()`, `setHeight()`, `upBy()`, `downBy()`, `down()`, `up()`. As these methods do not use any new techniques, we will not explore them further. However, you may wish to study their code for your own interest. More significant are the two methods which were defined in the `Frog` class, but which require slightly different behaviour for `HoverFrog` objects: `home()` and `toString()`. Defining a method with the same name and arguments (the same signature) as a method in a superclass is called **overriding**.

Java does not insist on instance variables being declared at the start of a class definition, but it's a common convention to put them there.

How the method `home()` for `HoverFrog` is implemented

You have seen in activities in earlier units that the method `home()` for the `HoverFrog` class does what the method `home()` in `Frog` does, but in addition it sets the height of its receiver to 0. The method *could* therefore have been written as follows:

```
public void home()
{
    this.setPosition(1);
    this.setHeight(0);
}
```

Although this would work perfectly well, Java provides a more elegant way of including the behaviour of a superclass in a method of its subclass, and one which better expresses the fact that a method is extending the behaviour of an existing, inherited method from the superclass. The first line of the method body above, `this.setPosition(1)`, is in fact exactly the same as the code in the method body for the method `home()` of `Frog`. We can therefore replace this line by the statement:

```
super.home();
```

When the method `home()` for `HoverFrog` is executed, this line will result in the execution of the method `home()` defined in the `Frog` class.

So the method `home()` for `HoverFrog` (excluding the initial comment) has been written as:

```
public void home()
{
    super.home();
    this.setHeight(0);
}
```

More generally, like the pseudo-variable `this`, the pseudo-variable `super` is used in a method to refer to the receiver, so that an object can send a message to itself. However, while the use of `this` causes the Java Virtual Machine (JVM) to start its search for the method corresponding to the message in the class of the receiver, the use of `super` causes the JVM to start its search for the method in the superclass of the class containing the method in which `super` appears. Not as you might think in the superclass of the receiver.

This means that although the use of `super` often leads to the invocation of the method defined for the direct superclass of the receiver, this is not always the case, as demonstrated by the following example.

Consider the following class hierarchy:

```
ClassA
  ClassB
    ClassC
```

ClassA and ClassB each have a method called `methodA()`, defined as follows:

```
/** method for ClassA */
public int methodA()
{
    return 1;
}

/** method for classB */
public int methodA()
{
    return 2;
}
```

The example has been kept deliberately simple so that you can concentrate on the behaviour of `super` rather than the behaviour of the methods, which is trivial.

ClassB also has a method, `methodB()`, defined as follows:

```
public int methodB()
{
    return super.methodA();
}
```

Now consider a ClassC which is a subclass of ClassB and which does not override either `methodA()` or `methodB()`.

- ▶ When an instance of class ClassC receives a message `methodA()`, the result returned is 2. As there is no `methodA()` defined for ClassC the method defined for the receiver is the definition of ClassB, which returns 2.
- ▶ When an instance of class ClassC receives a message `methodB()`, the result returned is 1. The use of the keyword `super` in the definition of `methodB()` causes the execution of `methodA()` *from the superclass of the class in which `methodB()` is defined* (i.e. ClassA), not from the superclass of the receiver (which would be ClassB).

In the definition of `home()` the use of `super` has not saved us much work, as the code for the superclass method which it uses is very simple. Nevertheless it makes clear the inheritance of behaviour from the superclass, and reduces the likelihood of introducing errors which could result from rewriting code already implemented for `Frog`. Furthermore, if the home position of frogs were to be changed, say to 3, then this new home position would automatically take effect for `hoverfrogs`, without the need for any changes to the code of the `HoverFrog` class. As you work through the course, you will see the power of `super` in supporting the reuse of much more complex methods.

Exercise 2

You might think that because `HoverFrog` inherits the method `home()` from `Frog` anyway, we could just write the method `home()` for `HoverFrog` as:

```
public void home()
{
    this.home();
    this.setHeight(0);
}
```

Can you see a problem with this?

For a method to be executed recursively without causing infinite recursion, it must include code to ensure that it will eventually stop calling itself.

Solution.....

If a `HoverFrog` object were to receive a message `home()` resulting in the execution of the method body above, it would start by sending a message `home()` to itself – which would involve sending a message `home()` to itself ... and so on! So the method `home()` would be executed again, and again, and again ... and execution would never get past the first line of the method. Java will continually attempt to execute `home()` methods until it runs out of memory. The process whereby a method invokes itself is known as **recursion**, or a **recursive method** call. There are circumstances in which recursion is a useful programming technique, but this is not one of them! By using the keyword `super`, to force invocation of the method `home()` from the superclass, recursive looping can be avoided.

How the method `toString()` in `HoverFrog` is implemented

As with `home()`, the method `toString()` in `HoverFrog` overrides the method in `Frog`. We will start by studying the method in the `Frog` class, which introduces some elements of Java that have not been discussed so far. Here is the method, the purpose of which is to provide a textual representation of a `Frog` object.

```
/**
 * Returns a string representation of the receiver
 */
public String toString()
{
    return "An instance of the class " + this.getClass().getName()
        + ":position " + this.getPosition()
        + ", colour " + this.getColour();
}
```

SAQ 2

The **method header** for `toString()` has four components: `public`, `String`, `toString` and `()`. Explain the purpose of each of these components.

ANSWER.....

- ▶ `public` is the access modifier which specifies which other classes have access to the method (this is explained in Subsection 1.5).
- ▶ `String` is the return type of the method.
- ▶ `toString` is the name of the method.
- ▶ `()` is the argument list, which is empty in this case.

Now let us look at the method body. We know that the method returns an instance of `String`, and you will recall from *Unit 3* that `+` is the concatenation operator for `String` objects. So all that the method `toString()` does is concatenate a number of strings to form a single string containing the information necessary to describe the receiver of the message, namely its class name and the names and values of its instance variables. Three of the strings to be included are literals; they are explicitly specified and will be the same every time the method code is executed. The other three components are strings that will result from message-send expressions. We will consider them one at a time.

```
this.getClass().getName()
```

The first thing that happens here is that the receiver of the message `toString()` (an instance of `Frog` or one of its subclasses) is sent a message `getClass()`. This is one of

the messages that `Frog` inherits from the class `Object`. The message returns an object which *represents* the run-time class of the receiver (not the actual class itself). This object is then sent a message `getName()`, which returns the name of the class as a `String`.

```
this.getPosition()
```

This message-send simply returns the position of its receiver. As you learnt in *Unit 3*, the concatenation operator used with a variable that holds a value of the primitive type `int` automatically causes the integer to be converted to an instance of `String`.

```
this.getColour();
```

In this case the colour of the receiver is initially returned. However when the `+` operator is used to concatenate an object with a string, the JVM will automatically invoke the method `toString()` for that object – in this case to return the colour as a `String`.

You are now going to define a method `toString()` for `HoverFrog`.

SAQ 3

What additional behaviour do you think will need to be added to the method `toString()` for `HoverFrog`?

ANSWER.....

It will need to retrieve the value of the `height` variable, add this to the string that would be returned by a `Frog` object with the same position and colour, and then return the complete string.

Exercise 3

Without looking at the code for the method `toString()` in the `HoverFrog` class, but using its method `home()` as a model, see if you can write (on paper) a method `toString()` for the `HoverFrog` class. Your method should include an initial comment and method header, and should make use of `super`. (This is a bit tricky, so do not spend too much time on it.)

Solution.....

Here is our method:

```
/**
 * Returns a string representation of the receiver
 */
public String toString()
{
    return super.toString() + ", height " + this.getHeight();
}
```

Your initial comment may not have been exactly the same as ours, but you should have used the comment style above, to ensure that the information is automatically picked up by Javadoc.

The method header will be exactly the same as for `Frog`. The message-send `super.toString()` causes the method `toString()` defined for the `Frog` class to be executed. This will return a string of the form: "An instance of the class `HoverFrog`: position `aNumber` colour `aColour`", where `aNumber` and `aColour` are replaced by the actual position and colour of the receiver. We then need to concatenate this string with a string providing information about the height of the receiver.

SAQ 4

Suppose we want to define a new public class, `ClassB`, as a subclass of an existing class, `ClassA`. What would be the Java class header for `ClassB`?

ANSWER.....

```
public class ClassB extends ClassA
```

SAQ 5

Suppose that `ClassA` and `ClassB` (as described in SAQ 4) both contain the definition of a method `doSomething()`. The code for `doSomething()` in `ClassB` is as follows.

```
public void doSomething()
{
    super.doSomething();           // line (1)
    this.doSomethingElse();        // line (2)
}
```

Suppose now that an instance of `ClassB`, referenced by `myB`, is sent the message `doSomething()`.

- (a) Which object(s) do `super` and `this` reference at run-time?
- (b) When the code `super.doSomething()` is executed, in which class would the JVM start looking for the method `doSomething()`?
- (c) When the code `this.doSomethingElse()` is executed, in which class would the JVM start looking for the method `doSomethingElse()`?

ANSWER.....

- (a) They both reference the receiver of the `doSomething()` message – namely the object referenced by `myB` (an instance of `ClassB`).
 - (b) It would bypass the method `doSomething()` in `ClassB` and start looking for a method `doSomething()` in `ClassA`, the superclass of the class implementing the method in which `super.doSomething()` appeared. (If there were no method `doSomething()` in `ClassA`, it would look in the superclass of `ClassA`, and so on up the class hierarchy).
 - (c) It would start looking in the class of the receiver, the object referenced by `myB`. If no such method existed in `ClassB()`, it would look in the definition of `ClassA()`, and so on up the class hierarchy.
-

Exercise 4

- (a) A method called `staggerRight()` is required for `Frog` objects. In response to a message `staggerRight()` the receiver should move right twice, jump, then move left one stone. The following two versions of `staggerRight()` are under consideration.

```
1 public void staggerRight()
  {
    this.setPosition(this.getPosition() + 2);
    this.jump();
    this.setPosition(this.getPosition() - 1);
  }
```

```

2 public void staggerRight()
  {
    this.right();
    this.right();
    this.jump();
    this.setPosition(this.getPosition() - 1);
  }

```

Which of (1) or (2) do you think best fits the specification for `staggerRight()`?

(b) `Righty` is a subclass of `Frog` in which the method `right()` is overridden as follows.

```

public void right()
{
    super.right();
    super.right();
}

```

For each of the implementations (1) and (2) of `staggerRight()` given above, what is the final position of `wittgenstein` after the following is evaluated?

```

Righty wittgenstein = new Righty();
wittgenstein.staggerRight();

```

In view of this answer, which of the two implementations of `staggerRight()` do you think defines the most 'appropriate' behaviour for instances of `Righty`?

Solution.....

- (a) Version two fits the specification best. When making decisions about the implementation of a method you need to bear in mind the possibility that the methods executed as part of the body of the method being written may be overridden by a subclass. The specification for the method `staggerRight()` distinguishes between 'moving right' and 'moving left *one stone*'. A subclass of `Frog` might override its inherited method `right()`, so that 'moving right' entailed something other than moving one stone, in which case it would be more appropriate for the subclass to use its own method `right()` rather than simply incrementing its position.
- (b) For implementation (1), the final position would be 2 (the instance would move two positions to the right, then one to the left); for implementation (2) it would be 4 (four positions to the right then one to the left). As argued in part (a), the second version of `staggerRight()` gives the most appropriate behaviour, as we would expect an instance of `Righty` to move two places to the right in response to each message `right()`.

The next section explores the role and definition of constructors, after which you will be in a position to create a new class of your own, but before leaving this section we briefly discuss the role of access modifiers in Java.

1.5 Access modifiers in Java

In the Java code that you have studied so far, all the instance variables and methods of classes have been declared as having a particular level of 'access', usually `public` or `private`. In Java, **access modifiers**, as these labels are called, define the extent to which program components – variables, methods, even classes – can be accessed by other parts of the program. Access control provides programmers with a tool for enforcing data hiding and so safeguarding the integrity of data. Java provides four

Access modifiers cannot be applied to **local variables** or arguments.

levels of access which can be applied to variables, methods and classes when they are defined:

- ▶ `private`: accessible only from within the instance methods of the class which defines them;
- ▶ `protected`: accessible from within the instance methods of the class which defines them; from instance methods of any subclasses of that class (regardless of where they are used) and from the instance methods of classes that are in the same package (library of related classes);
- ▶ `public`: accessible to all classes of objects in the system;
- ▶ if the programmer does not specify an access modifier the access defaults to 'package access' which provides access to the instance methods of the class that defines it and to the instance methods of the other classes in the package that contains that class, but not to subclasses of that class which are in a different package. If we were listing the levels of access in order from the most to the least restrictive, 'package access' would come between `private` and `protected`.

You may be puzzled by the fact that a class can be accessible to the class that defines it. This is because it is possible to define a class (known as an inner class) in another class. However this is beyond the scope of this course.

We will not be creating our own packages in M255, and will not need to deal with `protected` or package access in detail, though we will be using packages provided by the Java system. We are simply mentioning all four levels of access in case you encounter references to these in the Java documentation.

To use access modifiers effectively the designer of a program needs to have a very clear idea in advance of the role that different classes will play in the system, as restricted access may cause unanticipated problems later in the development process. For example, an instance of an unplanned subclass (or any other subclass for that matter!) will not be able to execute inherited methods declared as `private` in its superclass. In order to avoid problems of this nature as we build on our classes, in M255 we will *generally* be making all our methods `public`, at least in the early stages of the course.

However, you will have noticed that we have designated our instance variables as `private`. This does not prevent other classes from getting or changing their value, as we have provided publicly available getter and setter messages for each instance variable, but it does serve to protect their integrity. Suppose, for example, that we had made the instance variables of the `Frog` class `public`, thus allowing them to be accessed directly (without using the getter and setter methods). This would enable objects of other classes to change the value of a `Frog` object's `position` or `colour` variable without also informing any observing user interface that a change had occurred. This would result in a situation where a user interface no longer reflected the true state of the instance. By forcing users to modify the instance variables of objects via their public setter methods, we ensure that this access is properly managed.

Another reason for restricting access to methods and variables is simply to hide them away. It may be that a method is needed only to carry out a subtask for some other method in the same class, such as performing a calculation, or checking some user input against details held on file. There is no reason for objects of other classes to access such methods, which are sometimes referred to as **helper methods**, and it is safer if they are prohibited from doing so. If objects of other classes have access to unnecessary details of implementation there is a temptation for these classes to be written in an implementation-dependent way, i.e. in a way which relies on this detail. This imposes constraints on the original class, which cannot be modified without the risk of causing dependent classes to stop working.

2 Constructors

Unit 4 introduced you to constructors as a way of initialising newly created objects. In this short section we explain a little more about the role and implementation of constructors, so as to equip you with the tools and understanding you will need to write constructors of your own.

You will recall from earlier units that a new instance of a class can be created and initialised by invoking the constructor for the class, preceded by the Java keyword `new`. For example, to create a newly initialised instance of the `Frog` class referenced by the variable `aFrog`, we would write:

```
Frog aFrog = new Frog();
```

As explained in *Unit 3*, the operator `new` first creates a `Frog` object and the code of the constructor `Frog()` is then executed to set the instance variables of the new `Frog` object to their initial values (1 and `OUColour.GREEN`).

When Java encounters `new` it expects a constructor to follow and constructors can only be used in conjunction with the operator `new`. A newly created instance of a class does not necessarily need to be assigned to a reference variable, of course; it can also be used as a message argument, or in any other situation where an object of that type is required. In the following message-send, for example, an existing instance of `Frog`, referenced by the variable `kermit`, has its `colour` instance variable set to the same value as that of a newly initialised `Frog` object.

```
kermit.sameColourAs(new Frog());
```

2.1 Implementation of constructors and the role of `super()`

All classes must have at least one constructor, though they may define more than one, as you will discover later in the unit. So how do you write the code for a constructor? The constructor for the `Frog` class, which is reproduced below, will serve as an example.

```
/**
 * Constructor for objects of class Frog which initialises
 * colour to green and position to 1
 */
public Frog()
{
    super();
    this.colour = OUColour.GREEN;
    this.position = 1;
}
```

At first glance, the code for a constructor looks much like that for a method, but the role of a constructor is rather different. Most significantly, the execution of a constructor does not result from a message being sent and, as you have already seen, a constructor must always be preceded by the keyword `new`. The syntax for a constructor differs from that of a method in a number of ways.

Exercise 5

Study the header of the constructor for the `Frog` class and identify three ways in which it differs from all the method headers that you have seen so far.

Solution.....

- ▶ It has the same name as the class.
 - ▶ The name of the constructor starts with an upper-case letter, unlike method names, which conventionally start with a lower-case letter. (This is a consequence of it having the same name as the class, of course, as class names start with upper-case letters!)
 - ▶ It has no return type.
-

These three features are characteristic of all constructors.

Now let us explore the constructor body. As explained in *Unit 4* the second and third lines of code initialise the instance variables defined for the newly created object. `Frog` objects have two instance variables, `colour` and `position`, and these are set to `OUColour.GREEN` and `1`, respectively. The first line of code is:

```
super();
```

You saw earlier in the unit that the keyword `super` in a method body represents the receiver of the message which follows it, as in the example `super.home()`. But in the constructor code no message name is given, and `super` is followed directly by an argument list `()`, albeit an empty one in this case. So what is going on? In Java `super` has more than one purpose. Here it is being used as an instruction to execute the constructor with no arguments from the *direct* superclass of the class being defined, so that any instance variables inherited from the direct superclass are initialised. All constructors must start by executing a constructor from their direct superclass. If the programmer omits to include code to do this, Java will call the zero-argument constructor of the superclass by default. But we strongly encourage you to include explicit code, so as to remind yourself of the automatic **initialisation** of variables from the superclass.

As all constructors invoke the zero-argument constructor of their direct superclass, a superclass constructor will, in its turn, invoke the zero-argument constructor of *its* superclass, and so on up the hierarchy to `Object`, thus ensuring that instance variables inherited from both direct and indirect superclasses are initialised. The process whereby constructors use `super()` to invoke each other up the hierarchy is known as **constructor chaining**. After `super()` a constructor should include code to initialise the instance variables particular to the class being defined, or to initialise inherited instance variables in a way which differs from their initialisation in the superclass.

The constructor for the `Frog` class has no arguments, but many constructors do, as you will see when we write a new constructor for the `Account` class. In the case of the `Frog` constructor no arguments are needed, as all instances of the `Frog` class are initialised to have the same state.

We have said that if the programmer fails to write a constructor, Java will supply a default constructor at run-time, which will execute the constructor of the superclass. It will also set any additional instance variables to default values of the appropriate type, known as the **standard default values**. For primitive types the default values provided are `false` (for Booleans) and `0` or `0.0` (for numeric types); for object types Java assigns a standard default value of `null`, a Java keyword indicating that the variable does not currently hold a reference to any object. But default initialisation is rarely adequate. For

all practical purposes you should assume that a constructor must be provided by the programmer for every class.

However to aid the programmer, when you create a class in BlueJ, the class template includes a default constructor which simply invokes `super()`.

Exercise 6

Without looking at the code in BlueJ, write (on paper) a constructor for the `HoverFrog` class. Remember that `HoverFrog` instances are initialised in the same way as `Frog` instances, but that in addition their `height` variable is set to 0. You need not include an initial comment.

Solution.....

```
public HoverFrog()
{
    super();
    this.height = 0;
}
```

This is very similar to the constructor for `Frog()`. The call to `super()` here will cause the constructor for the `Frog` class to be executed, which will in turn execute the constructor for `OUAnimatedObject`, and so on up the class hierarchy.

You have now learnt everything you need in order to define a simple class as a subclass of an existing class. The box below gives an outline of what is required, using the `HoverFrog` class as an example.

Defining a subclass

Specify that the class is a subclass of an existing class using the Java keyword `extends`:

```
public class HoverFrog extends Frog
```

Declare any additional instance variables needed by the new class:

```
private int height;
```

Modify the constructor for the new class, provided by the class template, to initialise the instance variables:

```
public HoverFrog()
{
    super();
    this.setHeight(0);
}
```

Define any completely new methods for the class (i.e. methods which do not appear in the superclass).

For `HoverFrog` these are `getHeight()`, `setHeight()`, `down()`, `up()`, `upBy()`, `downBy()`.

We have omitted initial comments from the methods in this outline, but you should always include them when defining methods in BlueJ – not least so that their specification is picked up by Javadoc.

Define any methods that override inherited ones, i.e. methods which have the same name and arguments as methods in the superclass, but which implement different behaviour. These methods may (but do not always) invoke the method from the superclass, using the keyword `super`.

```
public void home()
{
    super.home();
    this.setHeight(0);
}

public String toString()
{
    return super.toString() + ", height " + this.getHeight();
}
```

2.2 Constructors and overloading

In Section 3 you will apply what you have learnt to implement a new class as a subclass of `Account`, but first we will revisit the constructor for the class `Account`, and introduce you to the important new concept of overloading.

Defining an additional constructor for `Account`

Here is the existing constructor for the class `Account`.

```
/**
 * Constructor for objects of class Account
 */
public Account()
{
    super(); // optional
    this.holder = "";
    this.number = "";
    this.balance = 0.0;
}
```

The constructor for `Account` is very similar to those for `Frog` and `HoverFrog`. As with those classes we have arranged for all the instances of the class to be initialised in an identical manner. All newly initialised instances of the `Frog` class have `colour` set to green and `position` set to 1. All newly initialised instances of the `Account` class have `balance` set to 0.0, and `holder` and `number` set to empty strings. Our initialisations for `Account` objects, although ensuring that values of appropriate types are provided, are not a very good representation of the behaviour of accounts in the real world. When opening a bank account, it is usual to allocate an account number and the name of the account holder at the time the account is created; very often an initial deposit is required too. These will clearly differ from account to account, so it would be convenient if we had a constructor which was flexible enough to use different initialisation values for different `Account` objects.

SAQ 6

From your study of methods, what do you think could be added to a constructor so that it could be used to initialise `Account` objects differently on different occasions?

ANSWER.....

Adding arguments would enable a constructor to use different initialisation values on different occasions.

Just as methods are often defined to have arguments, so are constructors. In fact, constructors without arguments are, if anything, the exception. Here is a constructor with arguments for the class `Account`.

```
/**
 * Constructor for objects of class Account which sets the values of
 * the holder, number and balance of the receiver to the
 * arguments holderName, accountNumber and anAmount respectively.
 */
public Account (String holderName, String accountNumber, double anAmount)
{
    super(); // optional
    this.holder = holderName;
    this.number = accountNumber;
    this.balance = anAmount;
}
```

In this version of the constructor the three instance variables are initialised to the values provided via the arguments.

Exercise 7

Why do you think that we have chosen to implement the `number` instance variable, representing the account number, as a string rather than an integer?

Solution.....

Although an account number is made up of numeric characters, its role is that of an identifier rather than a number. It is unlikely that it will be used in calculations, so it is more appropriate to implement it as a `String` object. When displaying or printing out numbers, most programming languages omit leading zeros, so that an account number such as 002356 stored as an `int` would be incorrectly displayed as 2356. This problem would not arise in the case of a `String`, where all the characters would be preserved.

Exercise 8

Write down a Java statement to declare a variable, `anAccount`, of type `Account` and assign to it a newly created instance of the `Account` class.

The new account should have holder set to "Josie Bloggs", number set to "121244" and balance set to 500.0.

Solution.....

```
Account anAccount = new Account("Josie Bloggs", "121244", 500.0);
```

Did you remember to include the quotation marks around the strings?

ACTIVITY 3

Open Unit6_Project_3 in BlueJ and make a copy by selecting Project | Save As from the menu bar and rename the project as Unit6_MyAccounts. The new project name should appear at the top of your BlueJ window. Open the editor on the `Account` class and add the new constructor (shown just below the answer to SAQ 6) directly after the existing constructor. (Do not delete the existing constructor.) Compile the class and then generate project documentation for the class to check that both constructors are listed.

Now open the OUWorkspace and use each constructor to create a number of instances of the `Account` class, assigned to variables of the appropriate type. Inspect the `Account` objects to see if their instance variables are initialised in the way you expect. Satisfy yourself that the accounts you have created with the new constructor respond correctly to other messages in the protocol of `Account`.

DISCUSSION OF ACTIVITY 3

You should have found that the modified `Account` class containing the two different constructor definitions compiled successfully, and that both constructors could be used to create `Account` objects which responded in the same way to all the messages in the protocol of the class.

If you plan to continue working without a break, keep the Unit6_MyAccounts project open in BlueJ, as you will be using it in the next section. If you could not get your project to work as expected, you can catch up in Activity 4 by using Unit6_Project_4 which includes both constructors.

It is the norm in Java (and some other object-oriented languages) to have more than one constructor, with different numbers of arguments – none, one, two, three or more – or the same number of arguments but with different types. Programmers creating instances of the class can choose the constructor which best suits their purpose. Similarly it is possible for a class to have two or more methods with the same name but different numbers or types of argument. We have already seen an example of this in the `Frog` class, which has two `sameColourAs()` methods, with the following signatures:

```
sameColourAs(Frog)
sameColourAs(Toad)
```

The situation where a class has two or more constructors or methods with the same name but different signatures, is known as **overloading**. In such a situation, it is the number and types of the actual arguments provided which determines which constructor or method is executed. You will learn more about overloading in *Unit 7*.

As you have already learnt, the combination of a method's name and the number and types of its arguments is known as its signature. For a given class, a method can be uniquely identified by its signature.

3

A subclass for Account

Now that you have seen what is involved in defining a simple subclass, and in writing constructors, we can return to the class `Account` for which you are going to define a subclass called `CurrentAccount`. Current accounts will allow their holders to be overdrawn up to a specified credit limit, and will have a PIN number for security. The class will have all the instance variables and methods of the `Account` class, but will have two additional instance variables (representing the credit limit and the PIN number), some additional behaviour, and one or more methods that behave differently from the method with the same name in the `Account` class. This section will guide you through the implementation of the new class by way of a number of practical activities. Writing the `CurrentAccount` class will be a fairly lengthy process and you may wish to break off from your studies between activities or during a particular activity. If you do this, make sure that you save the work you have completed so far, by selecting Project | Save from the BlueJ menu. If, at the end of a particular activity, you plan to continue working through the section, you should keep your current project open in BlueJ.

3.1 Creating a new class in BlueJ

ACTIVITY 4

If you successfully completed Activity 3, open your `Unit6_MyAccounts` project in BlueJ. (Otherwise, use `Unit6_Project_4` which contains the code from the previous activity.)

- 1 Regenerate the documentation for the project and scroll through the Constructor and Method summaries for the class `Account` to remind yourself of its constructors and methods. If you are using `Unit6_MyAccounts` you should ensure that you have added the additional constructor discussed in Subsection 2.2.
- 2 Select the main BlueJ window and click the New Class button. A window will open with a number of radio buttons and a prompt for the name of the class. Enter `CurrentAccount` into the text box, make sure that the top radio button (Class) is selected, and click Ok. A rectangle representing the new class should appear in the main BlueJ window.
- 3 Double-click on the `CurrentAccount` icon to open the editor. You will see that the editor provides you with a template for a new class. Replace the commented section at the top of the page with a short description of the class, together with your name and the version number and date. Complete the class header:

```
public class CurrentAccount // etc.
```

to indicate that this class will be a subclass of `Account`, and then click the compile button at the top of the window. If the class compiles correctly, you will get a message in the small pane at the bottom of the editor window which reads: `Class compiled – no syntax errors`, confirming that you have not introduced any errors!

- 4 Regenerate the project documentation and check that all the methods inherited from `Account` are available.

DISCUSSION OF ACTIVITY 4

The required class header is:

```
public class CurrentAccount extends Account
```

It must be exactly like this to ensure that your class inherits all the necessary instance variables and methods from `Account`, so even if your class compiled with a different header go back and correct it. Once you have the correct class header, you should be able to see from the project documentation that `CurrentAccount` inherits all the behaviour of `Account`.

If you need to take a break at this stage, make sure that you save your project by selecting Project | Save from the BlueJ menu bar before leaving your computer or closing down BlueJ.

We are now going to lead you through the completion of the class. After completing each stage you should compile the class to ensure that your code is syntactically correct and consistent so far. You might also want to create one or more instances of `CurrentAccount` in the OUWorkspace to check that the code executes without problems and implements the intended behaviour. Remember that syntactically correct code does not guarantee problem-free execution and that code can execute correctly without achieving the expected results.

3.2 Declaring instance variables and defining accessor methods

ACTIVITY 5

If you successfully completed Activity 4 using your Unit6_MyAccounts project, carry on using that, otherwise open Unit6_Project_5 to which we have added the code from the previous activity.

- 1 Here is a declaration for the additional instance variables needed for instances of the `CurrentAccount` class. Add these to the class in the place specified in the template. Then check that the class compiles.

```
private double creditLimit;
private String pinNo;
```

- 2 Now write getter and setter methods for both new instance variables to match the initial comments and method headers given below. Add the methods, including the comments, to your class, checking that it compiles without errors after the addition of each new method.

```
/**
 * Returns the creditLimit of the receiver
 */
public double getCreditLimit()

/**
 * Sets the new creditLimit of the receiver to the
 * argument aLimit
 */
public void setCreditLimit(double aLimit)
```

```

/**
 * Returns the pinNo of the receiver
 */
public String getPinNo()

/**
 * Sets the new pinNo of the receiver to the argument aPin
 */
public void setPinNo(String aPin)

```

DISCUSSION OF ACTIVITY 5

- 1 As usual, we have made the instance variables `private` so that instances of other classes can access them only by using publicly available messages. We need only define the two variables specific to the `CurrentAccount` class as the other three – holder, number and balance – will be inherited from the `Account` class.
- 2 Here is our code for the getter and setter methods (excluding the initial comments):

```

public double getCreditLimit()
{
    return this.creditLimit;
}

public void setCreditLimit(double aLimit)
{
    this.creditLimit = aLimit;
}

public String getPinNo()
{
    return this.pinNo;
}

public void setPinNo(String aPin)
{
    this.pinNo = aPin;
}

```

Your methods should have been very similar, if not identical, to ours. Note that these methods are for illustrative purposes only, in the real world getting and resetting a PIN number would require a number of security checks!

SAQ 7

In the `CurrentAccount` class we have declared the instance variable `pinNo` as being of type `String`. Can you think why we have done this rather than declaring it as type `int`?

ANSWER.....

In the real world, PIN numbers can begin with a zero, for example 0987. If we declared the instance variable `pinNo` to be an `int` rather than a `String`, instances of `CurrentAccount` would not be able to have PIN numbers beginning with zero.

3.3 Writing the constructors

When you create a new class, BlueJ provides a zero argument default constructor, consisting simply of the code `super()`.

SAQ 8

If a `CurrentAccount` object were created using this default constructor, what initial values would be assigned to the instance variables `holder`, `number`, `balance`, `creditLimit` and `pinNo`?

ANSWER.....

The inherited instance variables, `holder`, `number` and `balance` would be initialised to `""`, `""` and `0.0`, respectively, as for instances of `Account`. The two new instance variables would be initialised with the standard default values: `0.0` for `creditLimit` and `null` for `pinNo`. In the next activity you will modify the constructor for `CurrentAccount` to ensure that both the additional variables are correctly initialised.

ACTIVITY 6

Either using your `Unit6_MyAccounts` project, or `Unit6_Project_6` to which we have added the code from the previous activity, modify the constructor for the `CurrentAccount` class, to initialise `creditLimit` to `0.0` and `pinNo` to `"0000"`, after the call of `super()`.

Hint: You may wish to refer to the constructor for `HoverFrog` discussed in Subsection 2.1.

Check that your modified class compiles without errors.

DISCUSSION OF ACTIVITY 6

Here is our modified constructor:

```
/**
 * Constructor for objects of class CurrentAccount.
 */
public CurrentAccount()
{
    super();
    this.creditLimit = 0.0;
    this.pinNo = "0000";
}
```

We are using 'pattern' here in a very broad sense, applicable to all aspects of programming. The term is also used more specifically to refer to specific classes designed to address a particular kind of programming problem.

This uses the same **pattern** as that employed by the constructor for the `HoverFrog` class. Recognising and reusing patterns in the structure of code is an important part of programming, and is of particular value when you are learning to program.

SAQ 9

Now that you have modified the constructor, you can create properly initialised instances of `CurrentAccount` and explore their behaviour by sending them messages.

Write down all the messages to which you expect `CurrentAccount` instances to respond at this stage.

ANSWER.....

```
getHolder(), setHolder(), getNumber(), setNumber(), getBalance(),
setBalance(), getCreditLimit(), setCreditLimit(), getPinNo(), setPinNo(),
credit(), debit(), transfer().
```

At this stage instances of the class can respond to all the same messages as instances of its superclass. (It can also respond to messages for which methods are defined in its indirect superclass, `Object` – but we did not expect you to include these.) If you could not remember all the messages from the protocol of `Account`, you could have looked them up in the project documentation for `Account` or `CurrentAccount`.

ACTIVITY 7

Here is an additional constructor for the `CurrentAccount` class, similar to the second constructor defined for the `Account` class, but with two additional arguments representing the credit limit and the PIN number.

```
/**
 * Constructor for objects of class CurrentAccount, which
 * sets the values of holder, number, balance, creditLimit
 * and pinNo to the arguments holderName, accountNumber,
 * anAmount, aLimit and aPin respectively
 */
public CurrentAccount(String holderName, String accountNumber,
                      double anAmount, double aLimit, String aPin)
{
    super(holderName, accountNumber, anAmount);
    this.creditLimit = aLimit;
    this.pinNo = aPin;
}
```

Either using your `Unit6_MyAccounts` project, or `Unit6_Project_7` to which we have added the code from the previous activity, add this constructor to the `CurrentAccount` class directly after the first constructor and recompile the class.

You are now well on the way to completing your new class. To check that everything is going to plan, create a couple of new instances of `CurrentAccount` in the `OUWorkspace` (one using each constructor) and assign them to appropriately declared variables. Inspect the instances to check that their instance variables have been initialised in the way you expect. Then send them the messages listed in SAQ 9 to check that they all work.

Are there any messages that do not correctly implement the behaviour of current accounts, as briefly described in the opening paragraph of Section 3?

DISCUSSION OF ACTIVITY 7

This new constructor follows a similar pattern to the constructors you have seen so far. The first line of code is slightly different in that it causes the JVM to execute a superclass constructor with arguments. The three inherited instance variables are initialised by the superclass constructor, leaving initialisation of the remaining two variables to this constructor.

If you have followed all the stages above, and have not introduced any syntax errors into your class, you should find that you are able to send all the messages to your `CurrentAccount` objects without causing any errors. However, two of the messages do not work as required for current accounts. Because holders of current accounts are allowed to go overdrawn, the methods `debit()` and `transfer()` need to allow this. We will remedy this in the next subsection.

3.4 Modifying the behaviour of the `CurrentAccount` class

ACTIVITY 8

For this activity, use either your own `Unit6_MyAccounts` project, or `Unit6_Project_8` to which we have added the code from the previous activity

- 1 Holders of current accounts are allowed to withdraw up to the combined total of their balance and their credit limit. So, for example, if I have 350.50 in my current account, and a credit limit of 500.00, I can withdraw up to 850.50. Add a new method to your class, called `availableToSpend()`, which calculates and returns the maximum amount available for withdrawal from a current account. Here is the initial comment for the method.

```
/**
 * Calculates and returns the amount available to spend,
 * the total of the receiver's balance and creditLimit
 */
```

Recompile your class and test your method by sending some `availableToSpend()` messages to instances of `CurrentAccount` with different balances. Compiling and testing your code after each modification will make it much easier for you to identify any errors.

- 2 Once you are satisfied that your method `availableToSpend()` is working correctly, add to your class a `debit()` method which should have a single `double` argument. For current accounts, the amount that can be debited depends not just on the balance of the account, but also on the total amount available for withdrawal, so your method will need to execute `availableToSpend()` as part of its code. Your `debit()` method should return `true` or `false`, depending on the success of the transaction, and you should include an initial comment. Once your method compiles successfully, declare, in the `OUWorkspace`, a variable of type `CurrentAccount` and assign to it a `CurrentAccount` object. Test that `debit()` works correctly by sending `debit()` messages to the `CurrentAccount` object in situations where the amount to be debited is:

- ▶ less than the balance;
- ▶ greater than the balance but within the credit limit;
- ▶ equal to the combined balance and credit limit;
- ▶ greater than the combined balance and credit limit.

- 3 Now send some `transfer()` messages to transfer money between two current accounts and check that they work correctly in situations similar to those outlined in part 2. Can you explain why transfer messages work correctly, even though you have not overridden the method `transfer()` in `CurrentAccount`?

DISCUSSION OF ACTIVITY 8

- 1 Here is our method `availableToSpend()` (excluding the initial comment).

```
public double availableToSpend()
{
    return(this.getBalance() + this.getCreditLimit());
}
```

There are other versions that would work using direct access of instance variables, but that is discouraged in M255.

- 2 Here is our method `debit()` for `CurrentAccount`.

```
/**
 * If the amount available to spend (the balance and the creditLimit)
 * is greater than or equal to the argument anAmount, the balance of
 * the receiver is debited by anAmount and the method returns true.
 * If the amount available to spend is less than the argument
 * anAmount, the method simply returns false.
 */
public boolean debit(double anAmount)
{
    if (anAmount <= this.availableToSpend())
    {
        this.setBalance(this.getBalance() - anAmount);
        return true;
    }
    else
    {
        return false;
    }
}
```

This method is very similar to its counterpart in the `Account` class. The only difference is that the amount to be debited is compared with the total available for withdrawal, rather than just the balance.

Note this is an example of a method that overrides an inherited method without using `super` in its first line to cause the execution of the inherited method (i.e. `super.debit(anAmount)`). This is because we do not want the behaviour of the overriding method to include the behaviour of the method defined in the superclass.

In a real banking system it might be decided to make `availableToSpend()` a private helper method, accessible only to instances of `CurrentAccount`. As explained earlier we will be making most of our instance methods public, so as to allow more scope for extending our classes.

- 3 You should have found that `transfer()` messages now work correctly. This is because `transfer()` uses `debit()` to check whether there are sufficient funds for the transfer to take place. Now that `debit()` has been overridden and behaves appropriately for the `CurrentAccount` class there is no need to make any modification to `transfer()`.

Note that although the code for `transfer()` has been defined in the `Account` class, when a message `transfer()` is sent to an instance of `CurrentAccount`, the code `this.debit(anAmount)` that appears in the method `transfer()` causes the `debit()` method in `CurrentAccount` to be executed because the JVM will start the search in the class of the receiver of the message `debit()`, which is `CurrentAccount`.

You have now created a fully working `CurrentAccount` class. This may be the first class that you have ever written from scratch, and, if so, it has probably taken you some time to get all the code working. However, you should have noticed that we were able to save a lot of effort by building on the code that had already been written. In guiding you through the implementation of the class, we have encouraged you to make maximum use of existing methods from the `Account` class, and even where you have needed to write additional code, it has followed **patterns** that you have seen in other classes. This is a strategy that you should aim to adopt as you continue working through the course – reuse existing classes and methods where possible, copy or adapt well-tried patterns otherwise. It is very rare that you will need to work out how to solve a programming problem entirely from scratch.

3.5 Additional behaviour for `CurrentAccount`

Before leaving the `CurrentAccount` class we will discuss three further methods. These will provide additional functionality to instances of the class and give you the opportunity to revise some of the ideas you learnt earlier in the course. The first of these, `checkPin()`, is a method to check a PIN number entered by the account holder against the PIN stored in the account. (The way in which the PIN is entered need not concern us.) If the string entered by the user, which will be passed as an argument to the method, matches the stored PIN, the method returns `true`; otherwise it returns `false`.

SAQ 10

What should be the return type of the method `checkPin()`?

ANSWER.....

The method should return a value of the primitive type, `boolean`.

SAQ 11

(a) Suppose you create two `String` objects, `aString` and `bString`, as follows:

```
String aString = new String("hello");
String bString = new String("hello");
```

What values will be returned by each of the following two lines of code when they are executed?

```
aString.equals(bString);
aString == bString;
```

(b) Suppose now that you execute the following statement:

```
aString = bString;
```

What values will be returned by each of the following two lines of code when they are executed?

```
aString.equals(bString);
aString == bString;
```

ANSWER.....

- (a) The message expression `aString.equals(bString)` will return `true` because `aString` and `bString` reference objects of the same class with the same state. The message expression `aString == bString` will return `false` because `aString` and `bString` do not refer to the same instance of `String`, they have been created as distinct objects.
- (b) The message expression `aString.equals(bString)` will return `true` because `aString` and `bString` still refer to objects of the same class with the same state (actually it's the same object!). The message expression `aString == bString` will return `true` because we have set the two variables to refer to the same instance of `String`. When comparing two strings to see if they have the same state you should always use the method `equals()`.

ACTIVITY 9

Either using your `Unit6_MyAccounts` project or `Unit6_Project_9` to which we have added the code from the previous activity, write the method `checkPin()`. Here is the initial comment and method header.

```
/**
 * Returns true if the pinNo of the receiver matches the
 * argument aPin, false otherwise.
 */
public boolean checkPin(String aPin)
```

DISCUSSION OF ACTIVITY 9

Here is our code for the method `checkPin()`.

```
public boolean checkPin(String aPin)
{
    return this.getPinNo().equals(aPin);
}
```

As explained in *Unit 5* the role of `equals()` is to compare two objects and check that they are 'equivalent'. This is generally interpreted to mean that they are instances of the same class, with the same state (that is, with instance variables set to the same values or referencing objects with the same state). In the case of the method `checkPin()`, the message `equals()` will check that `pinNo` and `aPin` are of the same class (in this case `String`), that they have the same number of elements (in this case four) and that they contain the same characters in the same order. If all these conditions are met the method will return `true`, otherwise it will return `false`.

The method `equals()` should not be confused with the `==` operator, which, for objects, tests whether two variables reference the same object.

Our next method for the `CurrentAccount` class will display some of the details of a current account.

ACTIVITY 10

Either using your Unit6_MyAccounts project or Unit6_Project_10 to which we have added the code from the previous activity, write the method `displayDetails()` for `CurrentAccount`. The method should take no arguments and should return nothing. The body of the method should simply print to the Display Pane (using `System.out.println()`) the values of the receiver's `holder`, `number` and `balance` instance variables. Here is the initial comment and method header.

```
/**
 * Prints to the Display Pane the name, number and the balance
 * of the receiver.
 */
public void displayDetails()
```

Once you have got your method to compile, regenerate the project documentation for the project and check that it shows both the constructors and all the methods for `CurrentAccount`.

DISCUSSION OF ACTIVITY 10

Here is our version of the method `displayDetails()`.

```
public void displayDetails()
{
    System.out.println("Name: " + this.getHolder());
    System.out.println("Account No: " + this.getNumber());
    System.out.println("Balance: " + this.getBalance());
}
```

The method constructs the necessary output strings by using the concatenation operator. The methods `getHolder()` and `getNumber()` return strings. As with the primitive type `int`, the concatenation operator will cause Java to perform an automatic conversion of the `double` value returned by `getBalance()`, so no explicit type conversions are necessary.

In implementing the `checkPin()` method for `CurrentAccount` the code made use of an `equals()` message to test whether two `String` objects were equivalent. All objects inherit `equals()` from the class `Object`, however this inherited method just tests for object identity (i.e. returns `true` if the receiver and the argument are the *same* object), just like the `==` operator. In the `String` class this inherited method has been overridden to return `true` if the receiver and the argument have the same state, i.e. if they both hold the same characters, in the same order. In the final activity in this section you will define `equals()` methods for the classes `Account` and `CurrentAccount`. The next SAQ and activity will lead you through the implementation of this method for `Account`.

SAQ 12

Two instances of the class `Account` are referenced by `myAccount` and `yourAccount`.

- Write down a Java expression which will evaluate to `true` if the balances of the two accounts are the same, and `false` otherwise.
- Write down a Java expression which will evaluate to `true` if the two accounts have the same account number, and `false` otherwise.
- Write down a Java expression which will evaluate to `true` if the balances *and* the account numbers of the two accounts are the same, and `false` otherwise.

ANSWER.....

(a) `myAccount.getBalance() == yourAccount.getBalance()`

The values of any two primitive types (such as `double`) can be compared using the `==` operator.

(b) `myAccount.getNumber().equals(yourAccount.getNumber())`

As you saw in the previous subsection, the states of two `String` objects can be compared using the message `equals()`. In this case, an `equals()` message with a string argument that represents the account number of `yourAccount`, is sent to the string that represents the account number of `myAccount`.

(c) `myAccount.getBalance() == yourAccount.getBalance()
&& myAccount.getNumber().equals(yourAccount.getNumber())`

As you learnt in *Unit 3*, two Boolean expressions can be combined to create a compound expression using the logical operator `&&`.

The answers to SAQ 12 show you how the values of the `balance` and `number` instance variables of two `Account` objects can be compared for equality. When writing an `equals()` method for the `Account` class we would also need to compare the values of the receiver and argument's holder instance variable.

ACTIVITY 11

For this activity, use either your own `Unit6_MyAccounts` project or `Unit6_Project_11` to which we have added the code from the previous activity.

Open the `Account` class in the BlueJ editor.

Here is the initial comment and header for a public method `equals()` for the `Account` class.

```
/**
 * Returns true if receiver is equivalent to (has the same
 * state as) the argument anAccount, false otherwise.
 */
public boolean equals(Account anAccount)
```

Write the `equals()` method for the `Account` class using the expression given in part (c) of the answer to SAQ 12 as your starting point.

Once you have got the class to compile successfully, create some instances of `Account` in the OUWorkspace and check that your method works correctly.

DISCUSSION OF ACTIVITY 11

Here is our solution (excluding the initial comment).

```
public boolean equals(Account anAccount)
{
    return this.getBalance() == anAccount.getBalance()
        && this.getNumber().equals(anAccount.getNumber())
        && this.getHolder().equals(anAccount.getHolder());
}
```

The method body differs from the expression given in the answer to SAQ 12 in that we have added a check on the equality of the `holder` instance variables and returned the result of evaluating the complete expression.

We can now make use of the method `equals()` in the `Account` class defined in Activity 11 to define a method `equals()` for `CurrentAccount`.

ACTIVITY 12

For this activity, use either your own Unit6_MyAccounts project or Unit6_Project_12 to which we have added the code from Activity 11.

Open the `CurrentAccount` class in the BlueJ editor.

Here is the header for a method `equals()` for `CurrentAccount`.

```
public boolean equals(CurrentAccount anAccount)
```

Write the code for this method. You will first need to send an `equals()` message to `super`, with `anAccount` as the argument to check the inherited instance variables (`holder`, `number` and `balance`) for equality. Once that is done you can then compare the `pinNo` and `creditLimit` of the receiver and argument for equality.

Test your method in the OUWorkspace with a number of pairs of `CurrentAccount` objects.

DISCUSSION OF ACTIVITY 12

Here is our solution.

```
/**
 * Returns true if receiver is equivalent to (has the same
 * state as) the argument anAccount, false otherwise
 */
public boolean equals(CurrentAccount anAccount)
{
    return super.equals(anAccount)
        && this.getCreditLimit() == anAccount.getCreditLimit()
        && this.getPinNo().equals(anAccount.getPinNo());
}
```

In case you have had difficulty with this activity, we have provided a complete implementation of the class `CurrentAccount` in the project `Unit6_Project_12_Sol`.

4

Revisiting the amphibian hierarchy

When we discussed the amphibian classes in Section 1, we avoided any mention of the `Toad` class. Yet it is quite clear that `Toad` objects share many of the characteristics of `Frog` and `HoverFrog` objects. In particular, instances of `Toad` have exactly the same attributes (instance variables) as instances of `Frog` and share much of the same behaviour. Would it have been possible to implement `Toad`, like `HoverFrog`, as a subclass of `Frog`, thus saving ourselves some coding work, and making clear the similarities between the two classes? In this section we examine further the similarities and differences of the various amphibian classes with a view to incorporating the `Toad` class into the same hierarchy as `Frog` and `HoverFrog`. We will start by considering the question posed above: would it be possible to implement `Toad` as a subclass of `Frog`? Let us address this question by first reminding ourselves of the criteria by which we decide whether a new class should be implemented as a subclass of an existing class.

Generalising from the discussion of the `Frog` and `HoverFrog` classes in *Unit 2*, we can say that it is appropriate to develop a new class as a subclass of an existing one if the new class needs to have all the features of the existing class, but extends or modifies it in some way. It might add extra instance variables or methods, or redefine (override) one or more of the existing methods so that they respond in a different manner. (Of course, it could be a combination of such changes.)

The key point to note is that a subclass will inherit everything in its superclass – it can add things and it can change things but it cannot choose not to inherit a particular feature it does not require.

Both the inheritance relationships that we have already discussed in this unit, `Frog/HoverFrog` and `Account/CurrentAccount`, meet the criteria discussed above.

Exercise 9

Using the criteria outlined above, consider whether the `Frog` class is an appropriate superclass for the `Toad` class.

Solution.....

You should have concluded that this would not be appropriate. Although most of the criteria are met, instances of `Toad` class do not respond to the message `jump()`. As subclasses inherit all the methods from their superclasses, it would only be possible to create a subclass of `Frog` which could *not* respond to a `jump()` message if `jump()` was declared as `private` in the `Frog` class. However, if `jump()` was declared as `private`, `HoverFrog` objects would then lose the ability to respond to `jump()` messages!

4.1 Capturing common behaviour: abstract classes

So how else could we implement the two classes to capitalise on their common behaviour? In terms of the criteria to be considered, we *could* implement `Frog` as a subclass of `Toad`, adding the method `jump()` and overriding the methods `left()` and `right()`. But although our limited implementation of frogs and toads will allow us to do this, it would not be a very good solution. Intuitively we know that a frog is not a special kind of toad, any more than a toad is a special type of frog. The course team has deliberately limited the behaviour of both classes, so as to keep the examples simple. But as toads have warty skins, we could, for example, have provided our `Toad` class with an instance variable called `numberOfWarts`, together with some associated methods. We would then have been faced with a situation where each of the `Frog` and `Toad` classes had some feature not shared by the other class.

So although `Frog` and `Toad` classes have a huge amount in common, neither is really an appropriate subclass of the other. Frogs and toads are two specialised types of the same generic type, amphibian, rather than one of them being a specialised type of the other. In fact we have been referring to instances of the `Frog`, `Toad` and `HoverFrog` classes as amphibians since the start of the course. It seems that what is needed is some kind of `Amphibian` class from which the `Frog` and `Toad` classes could inherit the attributes and behaviour that they have in common. If we were to define `Amphibian` as a direct subclass of `OUAnimatedObject`, and both `Toad` and `Frog` as direct subclasses of `Amphibian`, we would have the inheritance hierarchy shown in Figure 4.

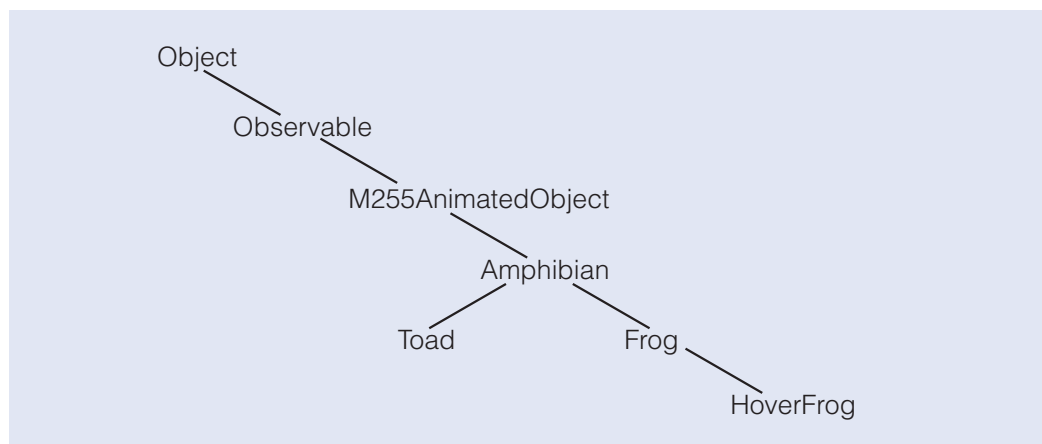


Figure 4 Suggested hierarchy for `Amphibian` objects, introducing a class `Amphibian`

In fact most of the classes shown in the animal classification in Figure 1 are analogous to abstract rather than concrete classes.

As amphibian is a generic term encompassing a number of particular types of amphibian, the concept of an `Amphibian` class is an abstraction, a convenient way of bundling together characteristics that are common to actual types of amphibian (in the same way that mammal is a way of encapsulating the common characteristics of a number of actual animal species). Most of the classes we use in our programs represent the part of the world we are interested in simulating – for example, `Frog`, `Account`. These are known as **concrete classes** and we create instances of these classes. We would not want to create any `Amphibian` objects, but it is often convenient to have abstract classes which will have no instances, but which specify the behaviour common to a number of concrete classes.

An **abstract class** defines a **common message protocol** and a common set of instance variables for its subclasses. Java allows us to designate a class as abstract, and the compiler will then bar us from creating any instances of that class.

In order to redesign our amphibian hierarchy to include an abstract `Amphibian` class, we must first identify the common message protocol and the common instance variables of the `Frog` and `Toad` classes.

ACTIVITY 13

Open `Unit6_Project_13` and open the editor on both the `Frog` class and the `Toad` class. Study the method summaries for the `Frog` and `Toad` classes, and identify the following.

- 1 The instance variables that are common to both the `Frog` and `Toad` classes.
- 2 The methods that have the same signatures and code for both classes.
- 3 The methods that have the same signatures but different code.

DISCUSSION OF ACTIVITY 13

- 1 There are two instance variables, `position` and `colour`, which are declared identically in both classes.
- 2 The methods with identical signatures and code are: `brown()`, `croak()`, `getColour()`, `getPosition()`, `green()`, `setColour()`, `sameColourAs()` (two methods), `setPosition()`, `toString()`.
- 3 The methods with identical signatures, but different code are: `home()`, `left()`, and `right()`.

The two instance variables common to both classes identified in part 1 of Activity 13 and all the methods identified in part 2 of Activity 13 can simply be moved out of `Frog` and `Toad` and into the abstract class, `Amphibian`, from where they will be inherited by the `Frog` and `Toad` classes. This process, not only makes for a better class hierarchy but also contributes to our general strategy of reuse.

What happens to the methods which have the same signatures but different code? We want both `Frog` and `Toad` to implement these methods, but each class to do so in its own way. To ensure that this happens, we must:

- ▶ define these methods as **abstract methods** in the abstract class;
- ▶ override them in each concrete class so as to provide the behaviour appropriate to that class.

You will see what all this looks like in our new class hierarchy, which we have implemented in project `Unit6_Project_14`.

ACTIVITY 14

Open `Unit6_Project_14` and have a look at the way the various classes are displayed in the main BlueJ window. You can see that the arrows between the classes reflect the modified inheritance hierarchy, and that the `Amphibian` class is labelled to show its abstract status. Now open the `Amphibian` class in the BlueJ editor to look at the code. There are a number of features of particular interest.

- ▶ The abstract nature of the class is specified in the class header using the Java keyword `abstract`:

```
public abstract class Amphibian extends OUAnimatedObject
```

- ▶ The instance variables and methods common to both direct subclasses (`Frog` and `Toad`) are defined in the abstract class. Note that we have only included a single method `sameColourAs()`, and that this has a formal argument of type `Amphibian`. We will return to this in Section 5.
- ▶ The constructor does nothing other than invoke the default constructor from its superclass. This does not mean that an abstract class could not contain code to initialise instance variables; it's just that, in this case, the instance variables in the direct subclasses are initialised differently.
- ▶ We have introduced three abstract methods: `left()`, `right()` and `home()`. As with the class header, these are labelled with the Java keyword `abstract`. They have no method bodies, as these will be supplied by the concrete subclasses. Note also the semicolon at the end of each method header. (We have defined our abstract methods directly after the instance variables for convenience, but they could be anywhere in the abstract class.)

Now briefly look at the code for the `Frog` and `Toad` classes. You will see that:

- ▶ both class headers have been modified to reflect the fact that the classes now extend the `Amphibian` class;
- ▶ the classes define only those methods that are declared as abstract in `Amphibian` and those that define behaviour that is particular to the class being defined (in this case, `jump()` in the `Frog` class).

There is one additional difference, which is not specifically related to the fact that `Amphibian` is an abstract class. Because the colour and instance variables are now declared in `Amphibian`, and are private to that class, we need to use their setter methods to initialise them in the constructors of the `Frog` and `Toad` classes.

All methods defined as abstract *must* be defined in every direct subclass of an abstract class, unless it too is an abstract class.

The implementation of `HoverFrog` is not affected by our new design. It is still a direct subclass of `Frog`, inheriting the same instance variables and methods as before, though some of these are now defined in `Amphibian()` rather than `Frog()`.

The redesign of a class hierarchy to better reflect the similarities and differences of behaviour in a number of interrelated classes is termed **refactoring**. The term is also used more generally for factoring methods to remove duplicate code (as you will see in *Unit 7*).

It is important to be aware that in Java a class declared as abstract can never have instances, even if it has a constructor (or constructors) containing explicit initialisation code. In order to create an object we must use a concrete, i.e. a non-abstract, subclass such as `Frog` or `Toad`. We describe this by saying that an abstract class cannot be **instantiated**. You will learn more about abstract classes in *Unit 7*.

SAQ 13

The variables `ferdie`, `tanya` and `horatio` reference instances of `Frog`, `Toad` and `HoverFrog`, respectively. For each of the following message-sends, state which class contains the code for the method which is executed at run-time.

- (a) `ferdie.right()`
- (b) `tanya.home()`
- (c) `horatio.left()`
- (d) `ferdie.setPosition()`
- (e) `horatio.toString()`
- (f) `ferdie.jump()`
- (g) `horatio.home()`

ANSWER.....

- (a) `Frog` – although specified in `Amphibian`, `right()` is an abstract method which is overridden in each of its direct concrete subclasses.
 - (b) `Toad` – `home()` is also specified as an abstract method in `Amphibian`, and so is overridden in the `Toad` class.
 - (c) `Frog` – `left()` is specified as abstract in `Amphibian`. It is overridden in the concrete class `Frog`, and needs no further overriding as instances of `HoverFrog` respond to `left()` messages in the same way as instances of `Frog`.
 - (d) `Amphibian` – `setPosition()` is defined as a concrete method in `Amphibian`. As all its concrete subclasses and indirect subclasses respond to a message `setPosition()` in the same way, there is no need for any of the subclasses to override it.
 - (e) `HoverFrog` – although the method `toString()` is defined in `Amphibian` and inherited unchanged by `Frog`, it needs to be overridden for `HoverFrog` to take into account the additional instance variable `height`.
 - (f) `Frog` – there is no method `jump()` in `Amphibian`.
 - (g) `HoverFrog` – the abstract method `home()` from `Amphibian` is overridden in `Frog()`, but must then be overridden again in `HoverFrog` to set the `height` instance variable to 0.
-

5

Subclassing, subtypes and substitution

We hope that this unit has convinced you how inheritance makes an important contribution to the production of Java code that is clear, concise and error free. We have concentrated on the reuse of existing methods in the definition of new classes. This short section, which consists mainly of practical activities, will enable you to explore the implications of subclassing on instance variables. It also provides an opportunity for you to experiment with instances of the classes that you have been studying in earlier sections.

General note on the activities in this section

All the activities use the same BlueJ project, Unit6_Project_15. For each activity we have provided a file containing code for you to execute in the OUWorkspace. You will find these files, which are named to match the associated activities (Unit6_Activity_15.txt, Unit6_Activity_16.txt, and so on), in the Unit6_Project_15 folder. To open the file for an activity you need to be in the OUWorkspace and then select Open from the File menu. The file browser will open, by default, in the folder containing the files for the current project. Select the required file and click Open to load the contents of the file into the OUWorkspace.

Each file contains a number of variable declarations, followed by a sequence of assignment and message-send statements. We have also added some comments, enclosed by `/*` and `*/`. Some of the statements in the code are purposefully invalid Java, which will produce error messages. You should therefore ensure that for each part of each activity you select and execute only the lines of code specified in this text.

You may also want to open a Graphical Display window, so that you can observe the behaviour of `Amphibian` objects created and manipulated by the code.

After completing each activity you should reset the OUWorkspace by selecting Reset OUWorkspace from the Action menu.

We suggest that you work through all the activities in the section in a single sitting, if possible.

ACTIVITY 15

Open Unit6_Project_15, which contains all the `Amphibian` and `Account` classes that you have studied or modified in the unit so far, and from the OUWorkspace open Unit6_Activity_15.txt which contains the code for this activity. Select and execute the first four lines of code which declare the variables. Then attempt to execute, one statement at a time, the lines of code which follow the comments labelled (a), (b), (c) and (d). After attempting to execute each statement make a note of the results, and of any error messages that appear in the Display Pane. The results will be clearer if you clear the Display Pane after each execution.

Once you have tested all four statements, study the statements and see if you can draw any general conclusions.

DISCUSSION OF ACTIVITY 15

- (a) The code executed and the variable `freda` now references a `HoverFrog` object. It is legal for an instance of `HoverFrog` to be assigned to a variable of type `Frog`.
- (b) The code caused an error. The error message indicated that the types on either side of the assignment operator are incompatible. Assigning an instance of `Frog` to a variable of type `HoverFrog` is illegal.
- (c) The code executed and the variable `myAccount` now references a `CurrentAccount` object. It is legal for an instance of `CurrentAccount` to be assigned to a variable of type `Account`.
- (d) The code caused an error. Assigning an instance of `Account` to a variable of type `CurrentAccount` is illegal.

Activity 15 has shown that:

- ▶ it is legal to assign an object to a variable declared to be of the type of its direct superclass;
- ▶ it is illegal to assign an object to a variable declared to be of the type of its direct subclass.

ACTIVITY 16

From the OUWorkspace open `Unit6_Activity_16.txt`. Attempt to execute the code in parts (a) to (f), one part at a time, and make a note of the results, including any error messages that appear in the Display Pane. Can you draw any general conclusions?

DISCUSSION OF ACTIVITY 16

- (a) The code executed and the variable `amy` appeared in the Variables pane. It is legal to declare a variable of type `Amphibian`, even though `Amphibian` is an abstract class.
- (b) The code caused an error. Java will not allow you to create an `Amphibian` object.
- (c) The code executed. It is legal to assign an instance of `Frog` to a variable declared as type `Amphibian`.
- (d) The code executed. It is legal to assign an instance of `Toad` to a variable declared as type `Amphibian`.
- (e) The code executed. It is legal to assign an instance of `HoverFrog` to a variable declared as type `Amphibian`.
- (f) The program would not execute due to incompatible types. It is illegal to assign an instance of `Toad` to a variable declared as type `Frog`.

Activity 16 has shown that:

- ▶ it is possible to declare variables of an abstract type, but it is not possible to create objects of an abstract class;
- ▶ it is legal to assign an object to a variable declared as the type of its superclass, even if that superclass is abstract;
- ▶ it is legal to assign an object to a variable declared as the type of one of its indirect superclasses as well as its direct superclass;
- ▶ it is illegal to assign an object to a variable of a type that is not the type of the object's class or not the type of a direct or indirect superclass, even if the types share a common superclass.

ACTIVITY 17

From the OUWorkspace open Unit6_Activity_17.txt. You might also want to open the Graphical Display so that you can observe the behaviour of the amphibian objects manipulated as part of the activity. Execute the first five code statements which declare and initialise some variables. Then attempt to execute the code in parts (a) to (c), one complete part at a time. Make a note of the results, including any error messages that appear in the Display Pane. Can you draw any general conclusions?

DISCUSSION OF ACTIVITY 17

- (a) The code executed and set the colour of the `HoverFrog` object referenced by `hercules` to blue, the colour of the `HoverFrog` object referenced by `houdini`. It is legal to provide an instance of `HoverFrog` as an argument to a method defined as having an argument of `Amphibian` class.
- (b) The code executed and set the colour of the `HoverFrog` object referenced by `hercules` to brown, the colour of the `Toad` object referenced by `titan`. It is legal to provide an instance of `Toad` as an actual argument to a method defined as having a formal argument of type `Amphibian`.
- (c) The code executed and successfully transferred 200 from the `Account` object referenced by `myAccount` to the `CurrentAccount` object referenced by `yourAccount`. It is legal to provide an instance of `CurrentAccount` as an actual argument to a method defined as having a formal argument of type `Account`.

From Activity 17 we can conclude that it is legal to supply as a method's actual argument an object which is a direct or indirect subclass of the type specified by the formal argument, even if that formal argument is of an abstract type.

This explains why it was possible to replace the two methods with the headers

```
public void sameColourAs(Frog aFrog)
```

and

```
public void sameColourAs(Toad aToad)
```

from the `Frog` and `Toad` classes by the single method

```
public void sameColourAs(Amphibian anAmphibian)
```

in the `Amphibian` class.

It would not be legal to supply, as a method's actual argument, an object whose class is a superclass of the formal argument. (Just as it is not legal to assign to a variable an object whose class is a superclass of the variable's declared type.)

ACTIVITY 18

Load the code from Unit6_Activity_18.txt into the OUWorkspace. Attempt to execute the code in parts (a) and (b), one complete part at a time, and make a note of the results, including any error messages that appear in the Display Pane. Can you draw any conclusions?

DISCUSSION OF ACTIVITY 18

- (a) The code executed and the colour of the `HoverFrog` object referenced by `freda` was successfully set to `red`.
- (b) This should have produced an error, but because of a peculiarity of the `OUWorkspace` the code has worked and the `HoverFrog` object referenced by `freda` now has `height` set to 4.

The `OUWorkspace`'s Java interpreter is erroneously using *the class of the object* referenced by the variable `freda` (which is `HoverFrog`) to determine what messages are valid, rather than *the declared type* of the variable `freda` (which is `Frog`). The BlueJ Java compiler correctly determines what messages are valid based on the type of the variable, therefore the compiler would reject the statement with the error message:

```
cannot find symbol - method setHeight()
```

as the method `setHeight()` is not part of the protocol of `Frog`. Try adding the following method to the `Frog` class to see for yourself:

```
public void test()
{
    Frog freda = new HoverFrog();
    freda.setHeight(4);
}
```

It is hoped that this peculiarity of the `OUWorkspace` can be fixed in a future release.

This activity has shown that although we can legally assign an object to a variable declared as one of its superclasses, the Java compiler will only allow that object to execute methods which are defined for the declared type of the variable.

Generalising from the results in Activities 15 to 18, we can conclude that where a Java program expects an instance of a particular class, it is legal to substitute an instance of its subclass. This is termed **substitutability** and an object of a class is said to be **substitutable** for an object of any of its superclasses. In *Unit 2* you learnt that methods with the same signature, to which more than one class can respond, are called **polymorphic methods**. What we have been describing in this section are **polymorphic variables**; that is, variables that can take on different forms at different times (the literal meaning of polymorphism is 'having many shapes'). In Java, most variables of object types are potentially polymorphic, as most classes can have subclasses. On the face of it, this property of variables may seem almost obvious. After all, instances of a class are, generally speaking, instances of their superclasses: a `HoverFrog` is a type of `Frog`, a `CurrentAccount` is a type of `Account`. However, although it is legal for a variable to reference an object of one of its subtypes, the compiler will not allow it to be used to execute a method (or access a variable) of the subclass that is not specified for objects of the declared type.

Substitutability is a powerful feature of inheritance which can contribute significantly to one of our core goals – reuse. For example, you saw how the method `sameColourAs()` defined with a formal argument of type `Amphibian` worked perfectly well with actual arguments which were `Toad` and `HoverFrog` objects. In Section 6 you will learn about a feature of Java that relies fundamentally on the concept of substitutability.

Not all classes in Java can have subclasses, but we will not pursue that here.

Exercise 10

The method `transfer()`, which is defined in the `Account` class, has the following signature:

```
public boolean transfer(Account toAccount, double anAmount)
```

This method is not overridden in `CurrentAccount`.

Suppose that `myAccount` and `yourAccount` reference instances of the classes `Account` and `CurrentAccount`, respectively. Part (c) of Activity 17 demonstrated that the property of substitutability would allow you to make a transfer such as the following:

```
myAccount.transfer(yourAccount, 500.00);
```

This means that you can use the method `transfer()` to make a transfer from an `Account` object to a `CurrentAccount` object. The method `transfer()` can also be used to make a transfer from an instance of `CurrentAccount` to an instance of `Account`, as in the following statement:

```
yourAccount.transfer(myAccount, 500.00);
```

Explain why this second statement is legal in Java.

Solution.....

`CurrentAccount` is a subclass of `Account` and therefore inherits its method `transfer()`. So it's perfectly all right to send a transfer message to an instance of `CurrentAccount`. The account argument provided is an instance of `Account`, as specified in the method signature.

6

An introduction to interfaces

So far in this unit we have concentrated on the specialisation of one class by another, with instance variables and methods being inherited or overridden. In this context, the instances of the related classes share instance variables and behaviour. But suppose we have a situation where a number of otherwise unrelated objects need to understand a shared set of messages, in circumstances where none of their actual instance variables or method code are the same. The sole common factor may be the group of messages that they must all respond to, and each class involved may already belong to a different inheritance hierarchy. We give a description of such a scenario.

Imagine that there is a class `MetOffice`, which simulates meteorological offices responsible for sending messages about the weather to objects of many different classes. The details of the class `MetOffice` need not concern you for the time being, but you can assume that it has no interest whatsoever in the objects to which it sends messages, or in any aspect of their behaviour other than their ability to understand messages about the weather. To keep things simple we will assume that there are only two such messages, `rain()` and `sun()`. What particular objects do in response to these messages will vary depending on their class.

Suppose also that there is a class `WeatherFrog`, which has the behaviour of a normal `Frog` and in addition can receive weather forecasts. The response of a `WeatherFrog` when sent the message `rain()` might be to tell its fellow frogs to take shelter. Instances of other classes might respond very differently to a message `rain()`. If we had a class `Daisy`, for example, its instances might respond by closing their petals; `Duck` objects might quack with enthusiasm, and so on.

The programming of the class `MetOffice` will be simplified greatly if it can assume that the objects that want to receive its messages (we could call them its clients), have `rain()` and `sun()` messages in their protocols. Making them all inherit from a common superclass would not be a good idea, because although they share a need to receive weather reports, in all other respects they may well be quite different. What sort of superclass would group ducks with daisies, for example? Obviously `WeatherFrog` should be a subclass of `Frog`, and this rules out inheritance from any other class as Java does not allow a class to inherit from more than one class.

The approach we have taken in this unit so far does not cater well for the situation we have just outlined, where nothing of the actual implementation may be held in common. Instead a somewhat different mechanism is required, provided by Java in the form of a programming structure called an **interface**.

Basically, an interface specifies a list of methods that a group of unrelated classes should implement, so that their instances can interact together by responding to a common subset of messages. Whoever draws up the list does not know, or need to know, how the instances of different classes will actually respond to the messages, so the interface provides only the method headers, not any method code – rather like the abstract methods you saw in Section 5.

The word ‘**client**’ is used in a number of similar, but not identical, ways in computing to indicate a software or hardware entity that uses a service provided by some other piece of software or hardware, which is known as a server.

The word ‘**interface**’ is used in a number of different contexts in programming. You have already come across the interface of a class – the messages to which its objects can respond – and graphical user interfaces. The Java keyword `interface` is yet another use, though it has some similarities with the use of the term interface to describe an object’s protocol.

6.1 Creating and implementing an interface type

Any class wishing to implement a particular interface must declare that it will implement that interface in its class header. It does this by using the Java keyword `implements`, followed by the name of the interface. You will shortly see some examples of this. Implementing an interface is a bit like making a contract – once a class signs up, it is committed to providing code for *all* the methods listed in the interface. If it were possible to leave any of them out, instances of the server class (in our example the `MetOffice` class) might broadcast a message to all its clients, only to find that it was not understood by all of them! This is precisely what an interface is designed to avoid.

To see how this works in practice the next two activities will lead you through the definition of a `WeatherClient` interface and the `WeatherFrog` class.

ACTIVITY 19

Open `Unit6_Project_16` in BlueJ. The project already includes the `Amphibian` classes, as well as a class `MetOffice`, a `Daisy` class (a very simple class) and an incomplete `WeatherFrog` class. The project will not yet compile as the `Daisy` and `MetOffice` classes both need to use the `WeatherClient` interface, which we have not yet implemented.

Click the New Class button and enter the name `WeatherClient` into the text box. Make sure that the radio button labelled `Interface` is checked, then click the button labelled `Ok`. The icon for the interface, suitably labelled, will appear in the main BlueJ window. (It might look as if it is already connected to other classes in the project, which is not the case. Drag it to a space of its own.) Now open the editor on `WeatherClient`. You will see that it already includes a header indicating that it's an interface, rather than a class. Copy the following two lines (which specify the headers of methods that classes which implement the interface must define) into the body of the interface.

```
public void rain();
public void sun();
```

You should now be able to compile the whole project, even though we have not yet implemented all the code we need.

DISCUSSION OF ACTIVITY 19

An interface is not a class and will have no instances of its own, and so it has no instance variables and no constructor. In fact it is an error to specify either instance variables or constructors for an interface.

To keep our example simple we have used methods with no arguments or return values. But it would be perfectly in order for methods in an interface to declare these if required.

Once the whole project has been compiled, the BlueJ window should show a dashed line with an inheritance-type arrow linking the class `Daisy` to `WeatherClient`. This style of arrow models implementation of an interface in UML.

Leave `Unit6_Project_16` open, as you will be using it for the next activity.

The Java syntax for an interface requires us to write each method header, followed by a semicolon. Since there is no body to the methods (these will be supplied by the client classes) no braces, {}, are needed.

Next we need to complete the code for the `MetOffice` client classes. Here is the code we have written for `Daisy` (a simple class which does nothing except listen for weather messages). `Daisy` is declared as implementing `WeatherClient`, which means it must provide code for the methods `rain()` and `sun()` specified in that interface. It needs no instance variables and its constructor will simply invoke the constructor from its superclass which is `Object`.

```
public class Daisy implements WeatherClient
{
    /**
     * Constructor for objects of class Daisy
     */
    public Daisy()
    {
        super();
    }

    /**
     * Causes the receiver to print "Opening" to the standard output
     */
    public void sun()
    {
        System.out.println("Opening");
    }

    /**
     * Causes the receiver to print "Closing" to the standard output
     */
    public void rain()
    {
        System.out.println("Closing");
    }
}
```

ACTIVITY 20

You are now going to complete the `WeatherFrog` class, which both extends its direct superclass (`Frog`) and simultaneously implements an interface. Here is its class header.

```
public class WeatherFrog extends Frog implements WeatherClient
```

Open `Unit6_Project_16` if it is not already open. Complete the class header for the `WeatherFrog` class as given above. Then try to compile the class. This will result in an error message telling you that you must override the method `sun()`. Remember that a class that implements an interface contracts to implement all its methods!

Now define the methods `sun()` and `rain()` for `WeatherFrog`, according to the specifications given in their initial comments in the implementation window. Neither method returns a value. Compile your code to check that you do not have any syntax errors. The class will inherit all its instance variables and methods from `Frog` and needs only a default constructor.

DISCUSSION OF ACTIVITY 20

Here is our code for the two methods (excluding the initial comments).

```
public void sun()
{
    this.setColour(OUColour.YELLOW);
    this.jump();
    this.jump();
    System.out.println("Sun on the way - you can come out now");
}
public void rain()
{
    this.setColour(OUColour.BLUE);
    this.croak();
    System.out.println("Rain on the way - under a lilypad everyone!");
}
```

Leave the project open if you plan to proceed to the next activity.

6.2 Using an interface type

So how do we use instances of classes that implement an interface? If we declare an instance of such a class, that instance will respond normally to all the messages in its protocol, including those that have been defined to meet the requirements of the interface. The following lines of code, for example, will have exactly the response you would expect:

```
WeatherFrog wendy = new WeatherFrog();
wendy.sun();
wendy.rain();
```

But this is not why we created an interface – we want to avoid the need for our `MetOffice` class to know anything about the classes of its clients. To see how this works you need to study the code for the `MetOffice` class. Here it is.

```
public class MetOffice
{
    // instance variables, all of type WeatherClient
    private WeatherClient client1;
    private WeatherClient client2;

    /**
     * Constructor for objects of class MetOffice.
     * Initialises the two instance variables to reference
     * the instances of client classes, given as the arguments.
     */
    public MetOffice(WeatherClient wClient1, WeatherClient wClient2)
    {
        super()
        this.client1 = wclient1;
        this.client2 = wclient2;
    }
}
```

```

/**
 * Causes the receiver to issue weather reports to client objects
 */
public void weatherReport()
{
    client1.sun();
    client1.rain();
    client2.sun();
    client2.rain();
}
}

```

In the `MetOffice` class we have declared an instance variable to reference each client object. Both these instance variables have been declared to be of the interface type `WeatherClient`. `MetOffice` does not care in the least what class of object these instance variables will reference at run-time, just that they implement the interface `WeatherClient`. (In a real-world application a single instance variable would probably be used, one that would reference some kind of collection object, such as an array, so it would not be necessary to specify the number of clients or give them individual variable names.) When an instance of `MetOffice` is created, its constructor will assign the objects that have been used as its arguments to the instance variables `client1` and `client2` (as you will see below). The `MetOffice` class also needs one or more methods which will enable its instances to respond to messages requesting weather reports. Our method `weatherReport()`, which is just for demonstration purposes, reports both sun and rain at the same time! In reality an instance of `MetOffice` would need to check the actual state of the weather and respond accordingly.

A collection object is one that can hold a number of other objects. You will learn about collection classes later in the course.

The mechanisms here are a little complex so we summarise the process so far.

When a `MetOffice` instance is created, the constructor is passed, as its actual arguments, two objects about which it knows nothing except that they are instances of classes which implement the `WeatherClient` interface. These objects are substituted for the formal arguments of the interface type.

`MetOffice` has an instance method `weatherReport()` which in turn sends the messages `sun()` and then `rain()` to each of the clients in turn. Although the actual class of these objects is unknown, they are guaranteed to understand both the messages, because they implement the `WeatherClient` interface.

In the next activity you will create an instance of the `MetOffice` class, with instances of `Daisy` and `WeatherFrog` as its clients, and request a weather report.

ACTIVITY 21

Open `Unit6_Project_16` and the `OUWorkspace`. You might also want to open a `Graphical Display` window to observe the behaviour of the `WeatherFrog` object.

First execute the following code to create instances of `WeatherFrog` and `Daisy` referenced respectively by the variables `wendy` and `dorian`:

```

WeatherFrog wendy = new WeatherFrog();
Daisy dorian = new Daisy();

```

Next create an instance of `MetOffice`, with its instance variables referencing the two clients created above with the code:

```

MetOffice bracknell = new MetOffice(wendy, dorian);

```

Finally send a message to the `MetOffice` object, requesting a weather report:

```
bracknell.weatherReport();
```

Observe the output in the Display Pane.

DISCUSSION OF ACTIVITY 21

You should have seen the following output in the Display Pane.

```
Sun on the way – you can come out now
Rain on the way – under a lilypad everyone!
Opening
Closing
```

If you have the Graphical Display window open, you will also see that the visual changes in the `WeatherFrog` object reflect its responses to the messages `sun()` and `rain()`. Each object has interpreted the messages `rain()` and `sun()` in its own way.

SAQ 14

What term can we use to describe methods with the same signature, but different behaviour when invoked on different classes?

ANSWER.....

The methods are polymorphic. In our example, the methods `sun()` and `rain()` (and their corresponding messages) are polymorphic.

In invoking the constructor for the `MetOffice` class we have been able to substitute actual arguments which are objects of the `Daisy` and `WeatherFrog` classes for formal arguments declared as type `WeatherClient`, demonstrating that method arguments of an interface type support **substitution**. Because they can refer to different types of object these method arguments (and indeed variables declared of some interface type) are also **polymorphic**. In fact, the concepts of **substitutability** and polymorphism are core to the whole idea of interfaces.

We could easily add any number of other classes which implement the `WeatherClient` interface, and the `MetOffice` would be able to communicate with instances of those classes in a similar manner. When programmers write interfaces, they generally will not have knowledge of all the classes that might eventually implement them. All kinds of implementing classes may be written in the future by other programmers. This creates no difficulty at all. Provided the implementing classes stick to the contract of the interface by supplying code for all the methods, everything will work perfectly.

Exercise 11

If in the future we added an extra method `windy()` to the `WeatherClient` interface, what problem would be caused?

Solution.....

It would break the contract with any class that already implements the original version of the interface. The class would no longer comply with the interface, because it would not implement `windy()`, and so it would suddenly stop working. An interface should never be enlarged, because doing so will make existing programs fail unexpectedly.

If you consult other sources of information on Java, you may find that interfaces are mentioned as a way of implementing multiple inheritance. Although it is possible for a class to implement one or more interfaces in addition to extending a superclass, interfaces are not classes and they provide a very limited form of ‘inheritance’ compared with the extension of a superclass by a subclass.

For example:

- ▶ the classes that implement an interface may have little in common with each other;
- ▶ the implementing classes inherit no instance variables and no behaviour, only a set of method headers.

The key feature of interfaces is their support for abstraction – the way in which they allow separation of the *specification* of behaviour (provided by the method headers) from its *implementation* (provided by its implementing classes). It is only the specification that is ‘inherited’. They therefore contribute very little to reuse, as the programmer still needs to fully implement all the methods.

In their support for abstraction, interfaces have some similarities with abstract classes, but, as you have seen, they do not operate as a conventional part of a class hierarchy, unlike abstract classes.

7

Summary

In this unit we have developed the ideas surrounding inheritance, which is a core feature of an object-oriented approach to software development. You have seen how subclassing can help manage complexity, and how inheritance can cut down on the work involved in writing and maintaining software, and help to guard against errors. When deciding on a class structure there are often choices to be made and we have provided you with some simple criteria to consider when deciding on the appropriateness of a subclass/superclass relationship. You have been introduced to Java interfaces, as a way of dealing with the situation where a number of disparate types of object need to respond to a common subset of messages.

Underlying both the effective use of inheritance and the implementation of interfaces is the notion that the class which inherits from a superclass or implements an interface has certain similarities with that superclass or interface. This leads to the important principle of substitutability, whereby an instance of a subclass, or a class implementing an interface, can be substituted for an instance of the superclass or the interface.

The unit has also discussed the role of constructors in the initialisation of variables

In the practical work for this unit, you have learnt about and practised some of the programming techniques involved in defining subclasses and in implementing and using a Java interface. You have learnt about overriding and overloading, and seen that both variables and methods in Java can be polymorphic. More generally the unit should have consolidated and developed your Java programming skills and given you the confidence to implement simple classes and methods.

A recurring theme of the unit has been the importance of reuse in software development by:

- ▶ creating classes as an extension of other classes;
- ▶ using the existing instance methods of a class, by sending the corresponding messages to `this` or `super`, when defining new methods;
- ▶ applying or adapting established coding and design patterns.

LEARNING OUTCOMES

Having studied this unit you should be able to:

- ▶ explain the principles of inheritance and its importance in object-oriented software development;
- ▶ list the criteria to be taken into consideration when deciding whether to implement a class as a subclass of some other class;
- ▶ describe the role of an abstract class in providing a common message protocol;
- ▶ explain the role of constructors in the creation and initialisation of objects;
- ▶ explain the difference between overriding and overloading;
- ▶ discuss briefly polymorphism in relation to variables and methods in Java;
- ▶ explain the principle of substitutability and describe some of its advantages;
- ▶ describe the role of Java interfaces;
- ▶ write simple methods, constructors and classes in Java, making appropriate use of language features designed to support inheritance;
- ▶ recognise opportunities for the reuse of existing methods and techniques in your programming;
- ▶ identify common errors in your code with the help of the error messages produced by BlueJ and the OUWorkspace;
- ▶ use the BlueJ environment to:
 - ▶ explore the Javadoc documentation for a class;
 - ▶ compile classes;
 - ▶ add classes to a project;
 - ▶ develop a class from scratch.

Glossary

abstract class A class that defines a **common message protocol** and common set of instance variables for its subclasses. In Java an abstract class cannot be instantiated.

abstract method A method declared as `abstract`. Abstract methods have no bodies. They must be implemented in any **concrete subclasses** of the class in which they are specified.

access modifier One of three Java keywords (`public`, `private`, `protected`) which specify the visibility of variables and methods.

cast A way of modifying the type of a variable or expression 'on the fly'.

class header The line in a class definition which provides crucial information such as its name, access modifier, name of a class from which it extends and name(s) of any interface(s) it implements. Example usage:

```
public class WeatherFrog extends Frog implements WeatherClient
```

client (class) In programming, an object of such a class uses a service provided by an object of some other class.

common message protocol A set of messages shared by a number of classes. Often used to describe the set of messages provided by an **abstract class** for its **subclasses**.

concrete class A class which is not **abstract**; a class for which instances can be created.

constructor chaining The process whereby constructors use `super ()` to invoke each other up the inheritance hierarchy.

data field A synonym for instance variable – and as you will learn in later units, it is also a synonym for class variable.

data member See **data field**.

direct subclass A class is a direct subclass of another class if it is directly below that class in the class hierarchy. In Java a class extends the class of which it is a direct **subclass**.

direct superclass A class is a direct superclass of another class if it is directly above it in the class hierarchy. In Java a class extends its direct superclass.

field See **data field**.

helper method A method which carries out some subsidiary task, such as a calculation, for another method. Helper methods would normally be `private`.

implement In software development, to write the program code for some task or specification.

implements A keyword in Java used in a class header to specify that the class implements a particular interface. Example usage:

```
public class SomeClass implements SomeInterface
```

indirect subclass A class is an indirect subclass of another class if it inherits from that class via one or more intermediate classes.

indirect superclass A class is an indirect superclass of another class, if it is above it in the class hierarchy but not directly above it.

inheritance A relationship between classes by which they are organised into a hierarchy. Classes lower in the hierarchy are said to *inherit* all the methods and variables from classes higher in the hierarchy (though they may also define additional methods and variables).

initialisation The setting of *variables* – both variables of *primitive types* and *instance variables* of *objects* – to appropriate values following their creation.

instance method The code that is executed as the result of a message being sent to an object.

instantiate Create an instance (of a class). In Java this must be a **concrete class**. **Abstract classes** and **interfaces** cannot be instantiated.

interface (Java specific) An interface specifies a list of methods that a group of unrelated classes should implement, so that their instances can interact together by responding to a common subset of messages. An interface only lists **method headers** (no method code), cannot declare any instance variables and cannot be **instantiated**. Classes implementing an interface must provide implementations for all the methods specified by that interface.

local variable A variable declared inside a method body, and whose scope is restricted to that method.

method header The line of code which precedes a method body and contains, at a minimum, the return type, name and argument list of the method. It may also include access modifiers and other information about the method.

new Java operator used in the creation of a new instance of a class.

null A special *value* in Java which indicates that a variable of an object type does not currently hold a reference to an object.

Object Top-level class in Java. Every class in Java is either a **direct** or an **indirect subclass** of `Object`.

overloading A method is said to be overloaded when there are other methods, defined in the same class, or inherited from some superclass, with the same name but a different *method signature*, i.e. different *types* and/or numbers or *arguments*.

overriding The process of redefining (replacing) a method inherited from a superclass so as to cause it to have different behaviour. A method which overrides a method from a superclass has the same *signature* as the superclass method.

pattern In programming, an established and well-tried approach, technique, algorithm or coding idiom that can be used as a model. (In object-oriented programming, pattern is also used to refer to particular named design structures involving a number of classes and modelling a solution to a common software design problem.)

polymorphic method A method with the same *signature* as some other method, but which defines different *behaviour*. For example the method `home()` defines different behaviour for `Frog`, `Toad` and `HoverFrog` objects.

polymorphic variable A variable which can *reference* objects of different *types*. In object-oriented languages, all variables declared of some object type or interface type are potentially polymorphic as they can be used to reference objects which are **subclasses** of that declared type or objects whose class implements that interface. See **substitution**.

private In the case of instance methods and instance variables, a Java access modifier restricting access to instance variables or methods to instances of the class that declares them.

public In the case of instance methods and instance variables, a Java access modifier which allows access to instance variables or methods from instances of any other class.

recursive method A method which calls itself as part of its method definition. This can lead to indefinite looping when an attempt is made to *execute* the method.

refactoring A technique whereby code is rewritten, without changing its overall effect, for the purpose of improving its design by removing code duplication. The term can be applied to big changes to code whereby a whole *class hierarchy* is altered, for example by the introduction of an abstract class, or to relatively small changes to a single class by factoring out duplicated code that appears in the class's methods into other **helper methods**.

standard default value The default values provided by Java to newly created variables.

subclass A subclass is any class which, when taking part in an inheritance relationship with another class, is the class to inherit functionality. In Java all classes except `Object` are subclasses of some other class.

subclassing A technique for defining a class as a **subclass** of an existing class.

substitutability The principle in object-oriented programming whereby, wherever the system expects an object of type X, an object of class Y can always be substituted instead, where class Y is a subclass of X, or where class Y implements an interface of type X. See **substitution**.

substitution The technique of providing an instance of one class in a situation where an instance of a different class is expected. In Java, an object can be assigned to a variable whose type has been declared as some superclass of the object, or which has been declared as an interface type which that object's class implements. Similarly, an object can be used as an actual argument to a method where the formal argument's type has been declared as some superclass of the object, or which has been declared as an interface type which that object's class implements.

super The pseudo-variable `super` is used within a method to refer to the receiver, so that an object can send a message to itself. However, while the use of `this` causes the JVM to start its search for the corresponding method in the class of the receiver, the use of `super` causes the JVM to start its search for the method in the superclass of the class containing the method in which `super` appears.

super() Used within a constructor to invoke the constructor without any arguments in the direct superclass.

this A pseudo-variable which, in a method or constructor, acts as a reference to the object which is executing that method or constructor, so that the object can send a message to itself.

Index

A

- abstract class 39
- abstract method 39
- access modifiers 17
- Account 22, 25
- availableToSpend() 30

C

- checkPin() 32
- class header 11
- classification 6–7
- common message protocol 39
- concrete class 38
- constructor 19
- constructor chaining 20
- CurrentAccount 25, 30

D

- data field 9
- debit() 31
- direct
 - subclass 10
 - superclass 10

E

- extends 11, 21

F

- fields 9

H

- helper method 18

I

- implements 48
- indirect
 - subclass 10
 - superclass 10

- inherit 7

- instantiate 40

- interface 47

J

- Javadoc 8

L

- local variable 17

M

- method header 14
- Method Summary 9

N

- new 19
- null 20

O

- Object 10
- overloading 24
- overriding 11

P

- package access 18
- pattern 28, 32
- polymorphic
 - method 45, 52
 - variable 45, 52
- private 18
- protected 18
- public 18

R

- recursion 14
- recursive method 14
- refactoring 40

S

- standard default value 20
- subclass 7, 10
- substitutability 45, 52
- substitution 52
- super 19

T

- this 12

